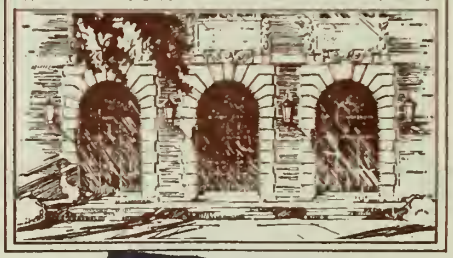


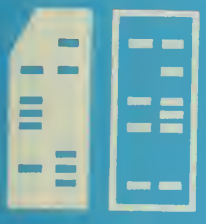
UIUCDCS-R-76-818

HEURISTICS THAT DYNAMICALLY ALTER DATA STRUCTURES TO DECREASE THEIR ACCESS TIME

by

James Richard Bitner

July, 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-76-818

HEURISTICS THAT DYNAMICALLY ALTER DATA STRUCTURES TO DECREASE THEIR ACCESS TIME*

by

James Richard Bitner

July 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the Department of Computer Science and in part by the National Science Foundation under Grant GJ-41538 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, 1976.

ACKNOWLEDGMENTS

I am very grateful to my advisor, Edward M. Reingold, for his many valuable suggestions during the preparation of this thesis, and to the other members of my exam committee: C. W. Gear, David Kuck, David Muller, and C. L. Liu. I thank D. L. Burkholder for his aid in proving the lemma on Page 109. I also wish to thank the typists at Techni-Typist for their fine job of typing this thesis and the National Science Foundation (Grants GJ-31222 and GJ-41538) and the Department of Computer Science for their financial support. Finally, I want to thank my family and friends for their understanding moral support.

TABLE OF CONTENTS

1. INTRODUCTION

Users of data structures frequently ignore some very valuable information: the number of times each key is requested. It is rarely the case that all keys are equally likely to be requested; some keys will be accessed frequently and others only rarely. Because we search for a key in a data structure by examining the locations in a certain order (which may depend on the results of previous key comparisons), the position that a given key occupies is important, and data will be retrieved much faster if high probability keys occur in the positions of the data structure that are searched first, that is, near the "top" of the structure.

There are various results in the case where the key request probabilities are known a priori, but this is seldom the case. If the probability distribution is not known beforehand, it must be observed as requests for keys are made. To take advantage of this information, the structure must be dynamically altered so that high probability elements rise to the "top" of the structure, and low probability elements sink to the "bottom." The purpose of this thesis is to evaluate and compare simple heuristics that perform this alteration.

Throughout this thesis, the expected access time will be used as the evaluation criterion for these heuristics. If c_i key comparisons are required to locate key k_i, which is requested with probability p_i, then the expected access time is defined by $\sum_{i=1}^{n} p_i c_i$. This is a good measure of the cost of accessing elements in the data structure

since it is equal to the two major costs, the expected number of comparisons and the expected number of links we must traverse (in a list or tree).

We will not consider the cost of performing the dynamic alteration because the rules we will look at are simple, and this cost should be quite small. Occasionally it is not, and these instances will be noted.

Depending on the exact probability distribution of key requests, substantial savings can be achieved if the arrangement of keys in the data structure is favorable. As an example, let us consider a linked list. If the order of the keys is random (each of the n! orderings is equally likely), the expected access time is $\frac{n+1}{2}$ since, on the average, we must search half-way down the list to find a given key. Note that this result holds for <u>any</u> probability distribution of key requests.

The optimal arrangement occurs when the elements of the list are in order of decreasing probability. The proof is simple: in any other ordering there must be a key that occurs before another keys which has higher probability. Interchanging these two keys results in a list of lower cost, and hence the original arrangement cannot be optimal.

An interesting generalization is given by Knuth [3, p. 400]. He supposes that the records are stored on tape and that the i^{th} has probability p_i and length L_i. It can then be proved that arranging the records in decreasing order of p_i/L_i will yield the optimal arrangement.

If the keys are optimally arranged, substantial decreases in access time can result as shown in Table 1.

Table 1

Comparison of Optimal and "Random" Costs

Distribution	Minimum Cost	Cost of Random Arrangement
$p_1 = 1$, $p_i = 0$ for $i > 1$	1	$\frac{n+1}{2}$
$p_i = r^i(\frac{1-r}{r-r^{n+1}})$, $r < 1$ (Geometric Distribution)	$\frac{1}{1-r} - \frac{nr^{n+1}}{r-r^{n+1}} \approx \frac{1}{1-r}$	$\frac{n+1}{2}$
$p_i = \frac{1}{iH_n}$ where $H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \ln n$ (Zipf's Law)	$\frac{n}{H_n} \approx \frac{n}{\ln n}$	$\frac{n+1}{2}$
Distribution of English Letters, see Kahn [5, p. 100]	7.5375	13.5
Distribution of the 50 most probable English words, see Kučera and Francis [6]	12.5718	25.5

The first distribution is rather unlikely, but points out that great decreases can occur. The minimum cost for the geometric distribution quickly approaches $\frac{1}{1-r}$, a constant, again, much smaller than $\frac{n+1}{2}$. If the distribution is in accordance with Zipf's Law, the random cost is

$\frac{\ln n}{2}$ times greater than the optimum. This can mean a factor of four or five for reasonably sized n. Both these formulas are easily derived by substituting the p_i into $\sum_{i=1}^{n} ip_i$(the optimal cost). For both English letters and the fifty most frequent English words, the random cost is approximately twice the optimum.

The costs of the heuristics described in the following section are shown to be at most twice the optimal cost. The exact cost, calculated for several distributions, is well under this bound. In fact, it is at most 38 percent larger than the optimum for these distributions.

Throughout this thesis, we will use ln x to denote the natural logarithm of x and log x to denote the base 2 log. Also, the following standard notations are used: $f(n) = O(g(n))$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is bounded, $f(n) = o(g(n))$ means this limit equals zero, and $f(n) = \Omega(g(n))$ means this limit is bounded, but <u>not</u> equal to zero.

2. LINKED LISTS

The first data structure we will study is the linked list. Throughout most of this chapter, we make the assumption that the request probabilities are constant with time and that the requests are independent. Because of these assumptions, we can model the behavior of an n-element list by a Markov chain* with n! states. Each state corresponds to one of the different orderings of the list. This model allows us to analyze the performance of the various rules.

This analysis will be from two different points of view. The first supposes that the number of key requests will be large compared to the number of elements in the list. In this case we are only concerned with the steady state of the Markov chain which tells us the asymptotic behavior.

The second point of view supposes there will be relatively few key requests. In this case, it is important how quickly the Markov chain approaches steady state from the initial distribution (in which each state is assumed to be equally likely). In general, the rate of convergence also indicates how well a rule will perform if the distribution varies with time. The more rapid the convergence, the more quickly a rule can adapt to a changing distribution.

---

*See the appendix for a summary of the important properties of Markov chains.

2.1 Asymptotic Results for Permutation Rules

We define a permutation rule as a set of n permutations $\{\tau_i, 1 \leq i \leq n\}$ of the integers $\{1,\ldots,n\}$. When the key in position i is requested, τ_i is used to reorder the elements of the list.

We will be primarily concerned with the following two permutation rules: The move to front rule moves the requested key to the top of the list, and the transposition rule transposes the requested key with the one above it. In both cases, if the requested key is already at the top of the list, no action is performed. We consider these rules because they are simple, and because the changes required on a linked list are cheaply executed. (If the list is sequentially allocated instead of linked, the move to front rule becomes very expensive to execute.)

Previous work in this area has been done [2 and 7] where the cost of the move to front rule is shown to be at most twice the optimal cost. Rivest [2] determined the steady state distribution of the transposition rule, and proved that it has lower asymptotic cost than the move to front rule. He also conjectured the transposition rule to be the optimal rule of all permutation rules. Yao [4] proved that the transposition rule is optimal assuming an optimal rule exists. The cost of the move to front rule has been determined [3,7,10] and analyzed by Knuth [3] in the case of Zipf's Law. Finally, Hendricks [8] has determined the steady state distribution for the move to front rule. These results will be proved in this section.

As noted in the appendix, both heuristics have steady state distributions and approach them from any initial distribution, and the asymptotic access time is the expected access time for the steady state distribution. We begin the analysis by determining the steady state distribution for the move to the front rule.

Theorem (Hendricks [8]): Consider any arrangement of n keys and label them $k_1,\ldots,k_n$ with probabilities $p_1,\ldots,p_n$ respectively. Using the move to the front rule, the steady state probability of this arrangement is

$$P(k_1,\ldots,k_n) = \frac{\prod_{i=1}^{n} p_i}{\prod_{i=1}^{n-1} \sum_{j=i+1}^{n} p_j}.$$

Proof: For the list to be in this ordering, k_i must have been requested more recently than k_{i+1}, $k_{i+2},\ldots k_n$, for $1 \le i \le n-1$. The probability that k_1, was requested after every other key is p_1. The probability that k_2 was requested last out of k_2, $k_3,\ldots,$ k_n is $\frac{p_2}{p_2 + p_3 +\ldots+p_n}$. In general, the probability that k_i was requested last out of k_i, $k_{i+1},\ldots,$ k_n is $\frac{p_i}{p_i+\ldots+p_n}$. Multiplying these probabilities gives the probability of the required sequence of key requests, which is

$$\frac{\prod_{i=1}^{n} p_i}{\prod_{i=2}^{n} \sum_{j=i}^{n} p_j} = \frac{\prod_{i=1}^{n} p_i}{\prod_{i=1}^{n-1} \sum_{j=i+1}^{n} p_j}$$

□

A more interesting statistic than the steady state distribution is the asymptotic access time (or "cost"). This is determined in the following theorem.

Theorem (Knuth [3, p. 399], Burville and Kingman [7] and McCabe [10]): Given keys $k_1, k_2, \ldots, k_n$ having probabilities $p_1, p_2, \ldots, p_n$, the asymptotic cost for a list ordered by the move to front rule is

$$1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + p_j}.$$

Proof: If we let ℓ_i be a random variable denoting the location of k_i,

$$E(\text{Cost}) = E\left(\sum_{i=1}^{n} p_i \ell_i\right) = \sum_{i=1}^{n} E(p_i \ell_i)$$

Since the expectation of a sum equals the sum of the expectations, and

$$= \sum_{i=1}^{n} p_i E(\ell_i)$$

since each p_i is a constant. To determine $E(\ell_i)$, define for $j \neq i$ random variables

$$A_{ji} = \begin{cases} 1 & \text{if } k_j \text{ is ahead of } k_i \text{ in the list} \\ 0 & \text{if not} \end{cases}$$

Since a given key's position is just one more than the number of keys ahead of it,

$$\ell_i = 1 + \sum_{j \neq i} A_{ji} \quad \text{and}$$

$$E(\ell_i) = 1 + \sum_{j \neq i} E(A_{ji})$$

But A_{ji} = 1 · Prob (k_j ahead of k_i) + 0 · Prob (k_j not ahead of k_i)

= Prob (k_j ahead of k_i).

Therefore

$$E(\ell_i) = 1 + \sum_{j \neq i} \text{Prob } (k_j \text{ ahead of } k_i)$$

and

$$E(\text{Cost}) = \sum_{i=1}^{n} p_i \left(1 + \sum_{j \neq i} \text{Prob } (k_j \text{ ahead of } k_i)\right)$$

$$E(\text{Cost}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \text{Prob } (k_j \text{ ahead of } k_i).$$

This last relation is very important and much use will be made of it.

Prob (k_j is ahead of k_i) is just the probability k_j was requested after k_i and therefore equals $\frac{p_j}{p_i+p_j}$. Substituting this into the cost formula gives

$$1 + \sum_{i=1} \sum_{j \neq i} \frac{p_i p_j}{p_i+p_j} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i+p_j}$$

Table 2.1.1 gives an indication of the magnitude of this cost. We can see from this table that the move to the front rule compares quite favorably with the optimum. The increase for the geometric distribution appears to reach a limit of 26.4%. Although the increase for Zipf's Law and large n cannot be seen from this table, Knuth [3, p. 399] has shown it to be approximately 38 percent for large n. The other two distributions considered have increases of 27.8 percent and 32.6 percent, which are also in the same range.

The steady state distribution of the transposition rule can also be determined.

Table 2.1.1

Asymptotic Cost of the Move to the Front Rule
Compared with the Optimal Cost

ENGLISH LETTERS		
OPTIMAL	MOVE TO FRONT RULE	% INCREASE
7.5375	9.6359	27.8

ENGLISH WORDS		
OPTIMAL	MOVE TO FRONT RULE	% INCREASE
12.5718	16.6667	32.6

GEOMETRIC DISTRIBUTION WITH N ELEMENTS

N	OPTIMAL	MOVE TO FRONT RULE	% INCREASE
3	1.5714	1.8000	14.5
4	1.7333	2.0607	18.9
5	1.8387	2.2392	21.8
6	1.9048	2.3550	23.6
7	1.9449	2.4270	24.8
8	1.9686	2.4704	25.5
9	1.9824	2.4958	25.9
10	1.9902	2.5105	26.1
11	1.9946	2.5188	26.3
12	1.9971	2.5234	26.4
13	1.9984	2.5260	26.4
14	1.9991	2.5274	26.4
15	1.9995	2.5281	26.4
16	1.9998	2.5285	26.4
17	1.9999	2.5287	26.4
18	1.9999	2.5289	26.4
19	2.0000	2.5289	26.4
20	2.0000	2.5290	26.4

ZIPF'S LAW WITH N ELEMENTS

N	OPTIMAL	MOVE TO FRONT RULE	% INCREASE
3	1.6364	1.8545	13.3
4	1.9200	2.2411	16.7
5	2.1898	2.6104	19.2
6	2.4490	2.9660	21.1
7	2.6997	3.3107	22.6
8	2.9435	3.6462	23.9
9	3.1814	3.9739	24.9
10	3.4142	4.2949	25.8
11	3.6425	4.6100	26.6
12	3.8670	4.9198	27.2
13	4.0879	5.2248	27.8
14	4.3056	5.5256	28.3
15	4.5205	5.8225	28.8
16	4.7327	6.1158	29.2
17	4.9425	6.4058	29.6
18	5.1501	6.6928	30.0
19	5.3555	6.9770	30.3
20	5.5590	7.2585	30.6

Theorem (Rivest [2]): Consider any arrangement of n keys and label them $k_1,\ldots,k_n$ with probabilities $p_1,\ldots,p_n$ respectively. Using the transposition rule, the steady state probability of this arrangement is

$$P(k_1,\ldots,k_n) = \frac{\prod_{i=1}^{n} p_i^{n-i}}{C}$$

where

$$C = \sum_{\text{all } \Pi} \prod_{i=1}^{n} p_{\Pi_i}^{n-i} \qquad \text{where } \Pi = (\Pi_1,\Pi_2,\ldots,\Pi_n)$$

is a permutation of $\{1,\ldots,n\}$.

Proof: We can easily verify that this is indeed a probability distribution since all terms are nonnegative and must sum to 1 (by definition of C).

To show it is the stationary distribution, we show $P(k_1,\ldots,k_n)$ satisfies the steady state equation:

$$P(k_1,\ldots,k_n) = p_1\, P(k_1,\ldots,k_n) + \sum_{i=1}^{n-1} p_i\, P(k_1,k_2,\ldots,k_{i+1},k_i,\ldots,k_n). \qquad (1)$$

From the definition of P we can see that

$$P(k_1,\ldots,k_{i+1},k_i,\ldots,k_n) = \frac{p_{i+1}}{p_i} \cdot P(k_1,\ldots,k_n)$$

Substituting this in (1) gives

$$\sum_{i=1}^{n-1} P(k_1,\ldots,k_i,k_{i+1},\ldots,k_n)\,\frac{p_{i+1}}{p_i}\,p_i + P(k_1,\ldots,k_n)p_1 =$$

$$P(k_1,\ldots,k_n)\Big(\sum_{i=1}^{n-1} p_{i+1} + p_1\Big) = P(k_1,\ldots,k_n).$$

Hence the distribution is the steady state distribution. □

From this distribution, we can determine the steady state cost by multiplying the cost of each state by its probability and then summing over all states. We have done this for Zipf's distribution and the results are summarized in Table 2.1.2. It is interesting to note that the difference from the optimum decreases as n increases (the difference increases for the move to the front rule). Also, the percentage increase is noticeably smaller than the 38 percent of the move to the front rule. In fact, Rivest [2] has shown this must hold for any distribution.

Table 2.1.2

Asymptotic Cost of the Transposition Rule
Compared with the Optimum

N	OPTIMAL	TRANSPOSITION RULE	%INCREASE
3	1.6364	1.8181	11.1
4	1.9200	2.1392	11.4
5	2.1898	2.4304	11.0
6	2.4490	2.7042	10.4
7	2.6997	2.9662	9.9
8	2.9435	3.2191	9.4
9	3.1814	3.4645	8.9

ASYMPTOTIC COST FOR ZIPF'S LAW WITH N ELEMENTS

Theorem (Rivest [2]): For any probability distribution, the cost of the transposition rule is less than or equal to that of the move to front rule.

Proof: Consider Prob(k_i is ahead of k_j) for the transposition rule. This is merely the sum of the probabilities of all states where k_i is ahead of k_j. These states have probabilities of the form $\frac{p_i^x p_j^y z}{C}$ (assume $p_i \geq p_j$), where $x > y$ and z is a product of powers of the other p_k's. We can pair each $p_i^x p_j^y z$ in the numerator with two terms ($p_i^x p_j^y z$ and $p_i^y p_j^x z$) in the denominator.

Since $\frac{p_i^x p_j^y z}{p_i^x p_j^y z + p_i^y p_j^x z} = \frac{p_i^{x-y}}{p_i^{x-y} + p_j^{x-y}}$, we get

$$p_i^x p_j^y z = \left(\frac{p_i^{x-y}}{p_i^{x-y} + p_j^{x-y}}\right)(p_i^x p_j^y z + p_i^y p_j^x z)$$

Since $1 \leq x-y$, $\frac{p_i}{p_i + p_j}(p_i^x p_j^y z + p_i^y p_j^x z) \leq p_i^x p_j^y z$. Summing over all states with $x > y$ and dividing by C gives

$$\frac{p_i}{p_i + p_j} \leq \text{Prob}(k_i \text{ ahead of } k_j)$$

Since $\frac{p_i}{p_i + p_j} = \text{Prob}(k_i \text{ ahead of } k_j)$ using the move to the front rule, and $E(\text{Cost}) = 1 + \sum_{i=1}^{n} p_i \sum_{i \neq j} \text{Prob}(k_i \text{ ahead of } k_j)$, the transposition rule is better than (or the same as) the move to the front rule. □

So the transposition rule has lower asymptotic cost than the move to front rule. Rivest [2] has conjectured that this result extends to all permutation rules, i.e. that the transposition rule is the optimal rule (has lowest asymptotic cost for any probability distribution) out of all permutation rules.

Intuitively, this conjecture is not surprising. The best we could possibly do (see Section 2.6) is to count the number of times each key has been accessed, and keep the keys ordered with respect to this count. The rule which most closely approximates this strategy is the transposition rule. We can also look at the situation in a different way: After a long time, the high probability keys are near the front of the list, and the low probability keys near the bottom. Occasionally, a low probability key will be accessed, and the move to the front rule will move it to the front of the list, increasing the expected cost since many high probability keys have moved down one position. The transposition rule does not do this, and it is difficult for the low probability keys to rise to high positions in the list.

While we cannot yet prove the transposition rule is optimal, it has been shown by Yao [4] that if an optimal permutation rule (optimal for all distributions) does exist, it must be the transposition rule. He does this by showing a particular distribution for which the transposition rule is optimal. Before discussing Yao's proof, we need a theorem by Rivest [2].

Theorem (Rivest [2]): An optimal permutation rule, $\{\tau_j, 1 \leq j \leq n\}$ (τ_j is used when the j^{th} key is requested) must have the property that each τ_j:

(i) leaves positions j+1 to n of the list fixed

(ii) if $j > 1$, moves the key in position j to some position $j' < j$.

Proof: Consider the probability distribution $p_i = 1/k$ for $1 \leq i \leq k$ and $p_i = 0$ for $k < i \leq n$, for some $k < n$. Any permutation rule satisfying (i) and (ii) above will have an asymptotic cost of (k+1)/2 since all of the keys with zero probability will move to the end of the list and stay there. Any permutation rule which violates (i) will occasionally move a key with zero probability in front of one with nonzero probability, and thus have greater asymptotic cost. Any permutation rule which satisfies (i) but not (ii) will not be able to move any keys out of positions j such that $\tau_j(j) = j$, so that the optimal ordering for this particular probability distribution cannot be reached, and, again, the asymptotic cost will be higher. □

Theorem (Yao [4]): Given a list of n elements with probability distribution $p_1 = 1-\varepsilon$ and $p_i = \frac{1-\varepsilon}{n-1}$, $2 \leq i \leq n$, there is an ε small enough such that the transposition rule is optimal for this distribution.

Proof: The Markov chain corresponding to this list has n distinct states, each one having k_1 (the key with probability $1-\varepsilon$) in a different position. Let q_i be the steady state probability that k_1 occupies

position i, using the transposition rule, and let r_i be this probability using the optimal rule. The transposition steady state satisfies:

$$q_1 = (1-\frac{\varepsilon}{n-1})\, q_1 + (1-\varepsilon)\, q_2$$

$$q_2 = \frac{\varepsilon}{n-1}\, q_1 + \frac{n-2}{n-1}\, \varepsilon\, q_2 + (1-\varepsilon)\, q_3$$

$$\vdots$$

$$q_{n-1} = \frac{\varepsilon}{n-1}\, q_{n-2} + \frac{n-2}{n-1}\, \varepsilon\, q_{n-1} + (1-\varepsilon)\, q_n$$

$$q_n = \frac{\varepsilon}{n-1}\, q_{n-1} + \varepsilon\, q_n$$

We solve this to get:

$$q_{j+1} = (\frac{1}{n-1}\ \frac{\varepsilon}{1-\varepsilon})^j\, q_1 \quad \text{for} \quad j = 1,\ldots,n-1$$

Since $\sum_{i=1}^{n} q_i = 1$, we obtain

$$q_1 = 1 + O(\varepsilon), \quad q_j = (\frac{\varepsilon}{n-1})^{j-1} + O(\varepsilon^j) \quad 2 \leq j \leq n.$$

From Rivest's Theorem, we know an optimal rule (if one exists) must have the form:

$$\tau_1 = \begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \end{pmatrix}$$

$$\tau_2 = \begin{pmatrix} 1 & 2 & 3 & \dots & n \\ 2 & 1 & 3 & \dots & n \end{pmatrix}$$

$$\tau_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & \dots & n \\ a_{31} & a_{32} & a_{33} & 4 & \dots & n \end{pmatrix}$$

$$\vdots$$

$$\tau_{n-1} = \begin{pmatrix} 1 & 2 & \dots & n-1 & n \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} & n \end{pmatrix}$$

$$\tau_n = \begin{pmatrix} 1 & 2 & \dots & n-1 & n \\ a_{n1} & a_{n2} & \cdots & a_{n-1,n} & a_{nn} \end{pmatrix}$$

The theorem now proceeds inductively in n-2 stages. At the k^{th} stage we will show:

1. τ_{k+2} is the same as the transposition rule.
2. $a_{ik} = k$ for $i \geq k + 2$.
3. $r_{k+1} = \frac{1}{n-1} \frac{\varepsilon}{1-\varepsilon} r_k = (\frac{\varepsilon}{n-1})^k + O(\varepsilon^{k+1})$.

Note that after stage n-2, we will have proven τ_i is the same as the transposition rule for i=1,...,n, and hence the theorem will be proved.

To begin the induction, we note that τ_1 and τ_2 are the same as the transposition rule by Rivest's Theorem. Hence condition 1 is

initially satisfied. Note that condition 2 vacuously holds if k=0. Finally, any rule satisfying the condition of Rivest's Theorem has $r_1=1$ when $\varepsilon=0$. Since $r_1 = \sum_{i=0}^{\infty} a_i\varepsilon_i$, if $\varepsilon=0$, $r_1=1=a_0$. Hence $r_1 = 1+0(\varepsilon)$ and condition 3 is initially satisfied. The proof for stage k proceeds as follows:

Let $N(i,j)$ be the number of ℓ such that $a_{\ell i}=j$. This is just the number of requests that cause the key in position i to move to position j. The r_i must satisfy their steady state equations. These give us the following bound

$$r_i \geq N(k,i) \cdot \frac{\varepsilon}{n-1} r_k \qquad k+1 \leq i \leq n$$

Here we have counted only those transitions from state k to state i and replaced the transition probability by a lower bound of $\frac{\varepsilon}{n-1}$. Summing these inequalities gives

$$r_{k+1} + \ldots + r_n \geq [\sum_{i=k+1}^{n} N(k,i)] \frac{\varepsilon}{n-1} r_k.$$

Set $\alpha = \sum_{i=k+1}^{n} N(k,i)$. Note that α is just the number of j's such that $a_{jk} > k$.

$$\geq \alpha(\frac{\varepsilon}{n-1})^{k+1} + 0(\varepsilon^{k+2}) \text{ by Condition 3}$$

So

$$r_{k+1} + \ldots + r_n = A(\frac{\varepsilon}{n-1})^{k+1} + 0(\varepsilon^{k+2}) \text{ for some } A \geq \alpha \qquad (1)$$

From this we conclude

$$(k+1)\, r_{k+1} + \ldots + n\, r_n \geq (k+1)\, A\left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2}) \qquad (2)$$

$$r_1 + \ldots + r_k = 1 - A\left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2}). \qquad (3)$$

We know that

$$(k+1)q_{k+1} + \ldots + n\, q_n = (k+1)\left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2}) \qquad (4)$$

since

$$q_{k+1} = \left(\frac{\varepsilon}{n-1}\right)^{k+1} \text{ and } q_i < O(\varepsilon^{k+1}) \text{ for } i > k+1$$

Subtracting (4) from (2) gives

$$(k+1)r_{k+1} + \ldots + n\, r_n - (k+1)q_{k+1} \ldots - n\, q_n \geq$$

$$(k+1)(A-1)\left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2}) \qquad (5)$$

Since

$$q_1 + \ldots + q_k = 1 - q_{k+1} - \ldots - q_n$$

we have

$$q_1 + \ldots + q_k = 1 - \left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2}) \qquad (6)$$

Now let $c = \frac{1}{n-1}\frac{\varepsilon}{1-\varepsilon}$. We have $q_i = c^{i-1}\, q_k, i \leq k$ and from property 3,

$$r_i = c^{i-1}\, r_1 \quad i \leq k \qquad (7)$$

Hence $q_1 + q_2 + \ldots + q_k = q_1(1+c+c^2+\ldots+c^{k-1}) = 1 - \left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2})$

$$q_1 = \frac{1 - \left(\frac{\varepsilon}{n-1}\right)^{k+1} + O(\varepsilon^{k+2})}{1 + c + c^2 + \ldots + c^{k-1}} \qquad (8)$$

Similarly

$$r_1 + \ldots + r_k = r_1(1+c+c^2+\ldots+c^{k-1}) \quad \text{using (3)},$$

$$= 1 - A(\tfrac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2}),$$

so

$$r_1 = \frac{1 - A(\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2})}{1 + c + c^2 + \ldots + c^{k-1}} \tag{9}$$

Now

$$q_1 + 2q_2 + \ldots + kq_k = q_1 + 2cq_1 + \ldots + kc^{k-1}\, q_1$$

$$= q_1(1+2c+\ldots kc^{k-1}).$$

Substituting for q_1 from (8)

$$= \left[1 - (\tfrac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2})\right]\left[\frac{1+2c+\ldots+kc^{k-1}}{1+c+\ldots+c^{k-1}}\right] \tag{10}$$

Similarly

$$r_1 + 2r_2 + \ldots + kq_k = q_1 + 2cq_1 + \ldots + kc^{k-1}\, q_1$$

$$[1 - A(\tfrac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2})]\left[\frac{1+2c+\ldots+kc^{k-1}}{1+c+\ldots+c^{k-1}}\right] \tag{11}$$

Subtracting (11) from (10) gives

$$q_1 + 2q_2 + \ldots + kq_k - r_1 - 2r_2 - \ldots - kr_k =$$

$$= [(A-1)(\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2})] \left[\frac{1+2c+\ldots+kc^{k-1}}{1+c+\ldots+c^{k-1}}\right]$$

$$< [(A-1)(\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2})] \left[\frac{k+kc+\ldots+kc^{k-1}}{1+c+\ldots+c^{k-1}}\right]$$

$$< k(A-1)(\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2}) \qquad (12)$$

Finally, subtracting (12) from (5) gives

$$r_1 + 2r_2 + \ldots + nr_n - q_1 - 2q_2 - \ldots - nq_n$$

$$< (A-1)(\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2}) \qquad (13)$$

But the left hand side of (13) is just the cost of the optimal rule minus the cost of the transposition rule. If $A > 1$, then the transposition has lower cost than the optimum rule, a contradiction. Hence $A \leqq 1$ and therefore $\alpha \leqq 1$. Recall that α is the number of $a_{ik} \neq k$. Since τ_{k+1} is the same as the transposition rule by Condition 1, we have $a_{k+1,k} = k+1$ and hence $\alpha \geqq 1$. Hence α must equal 1. Thus all other $a_{ik} \leqq k$ (else, $\alpha > 1$). If $a_{ik} < k$, Condition 2 will be violated since this value has already appeared in the permutation. Hence $a_{ik} = k$, proving Condition 2 for k. This determines the equation for r_{k-1}. If k=1,

$$r_1 = (1 - \frac{\varepsilon}{n-1})r_1 + (1-\varepsilon)r_2$$

$$r_2 = \frac{\varepsilon}{(1-\varepsilon)(n-1)} r_1 = \frac{\varepsilon}{n-1} + O(\varepsilon^2)$$

since $r_1 = 1+O(\varepsilon)$. If $k > 1$

$$r_k = \frac{\varepsilon}{n-1} r_{k-1} + \frac{n-2}{n-1} r_k + (1-\varepsilon) r_{k+1}$$

Solving this for r_{k+1} and substituting $(\frac{(n-1)(1-\varepsilon)}{\varepsilon}) r_k$ for r_{k-1} (Condition 3) gives $r_{k+1} = (\frac{\varepsilon}{(n-1)(1-\varepsilon)}) r_k$ and since $r_k = (\frac{\varepsilon}{n-1})^k + O(\varepsilon^{k+1})$ (again, Condition 3)

$$r_{k+1} = (\frac{\varepsilon}{n-1})^{k+1} + O(\varepsilon^{k+2}).$$

In either case, Condition 3 is proved. All that remains is to prove Condition 1. From Condition 2, we know $a_{k+2,i} = i$ for $i \leq k - 1$. We now know $a_{k+2,k} = k$, and hence τ_{k+2} is the same as the transposition rule, completing the induction for Condition 1 and proving the theorem. □

A final question is how far these rules can possibly be from the optimum. This is answered for the move to the front rule by Rivest [2] and Burville and Kingman [7]. If we assume $p_1 \geq p_2 \geq \ldots \geq p_n$ are the key request probabilities, then

$$\frac{\text{MTF Cost}}{\text{Opt Cost}} = \frac{1+2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{p_i p_j}{p_i+p_j}}{\sum_{i=1}^{n} i\, p_i} \leq \frac{1+2x}{1+x}$$

where $x = \sum_{j=1}^{n} p_j(j-1) \leq 2(1 - \frac{1}{n+1})$ since $x < \frac{n-1}{2}$.

Therefore, the move to the front rule never does more than twice the

work of the optimal ordering. The theorem also holds for the transposition rule as its cost is less than or equal to that of the move to front rule. This may be a significant savings, as remarked in the introduction.

In summary, the situation in the asymptotic case is quite clear: the transposition rule has asymptotic cost less than or equal to that of the move to front rule. Both rules compare quite favorably with the optimal cost. For the distributions we considered, the transposition rule was within 10 percent of the optimum and the move to front ranged from 25 percent to 38 percent. Finally, the cost of these rules is at most twice the optimal cost for any probability distribution.

2.2 Rate of Convergence

In the previous section, we only considered asymptotic behavior and found the move to the front rule inferior to the transposition rule. In this section we will consider how quickly the rules approach their asymptotes. We will find that the move to the front rule approaches its asymptote more quickly, and initially has a lower expected cost than the transposition rule.

The reason for this is clear. In the initial random ordering, many high probability elements are far down in the list. These must be brought to the front to reduce the cost. Obviously, the move to the front rule will do a better job here since these keys make large jumps and quickly rise to the top. The transposition rule allows keys to move only one step at a time, so the convergence should be rather slow.

When key k_i is requested, it moves up one position, decreasing the cost by p_i since we can locate k_i with one less compare, and increasing the cost by p_{i-1}, since key k_{i-1} (the key above k_i) moves down one position, resulting in a net decrease of $p_i - p_{i-1}$. If the p_i's are "close" in size, they are $O(\frac{1}{n})$, and this decrease is $O(\frac{1}{n})$, resulting in a very slow convergence. We would expect the move to the front rule to take $\Omega(n)$ time to get very close to steady state, assuming $\Omega(n)$ high probability keys. The transposition rule should require $\Omega(n^2)$ since each key must move $\Omega(n)$ steps to get near the top.

To begin the analysis, we determine the expected cost of the move to the front rule as a function of time.

Theorem: Given keys $k_1, k_2, \ldots, k_n$ having request probabilities $p_1, p_2, \ldots, p_n$, the expected cost of accessing a list being modified by the move to front rule after t requests is

$$1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} + \sum_{1 \leq i < j \leq n} \frac{(p_i - p_j)^2}{2(p_i + p_j)} (1 - p_i - p_j)^t$$

Proof: We begin by deriving Prob (k_i is ahead of k_j at time t). There are two different situations that could cause k_i to be ahead of k_j. First, neither k_i nor k_j was requested in t requests and k_i was initially ahead of k_j. The probability of this is $\frac{1}{2}(1 - p_i - p_j)^t$. Second, k_i's most recent request was at time $m \geq 1$, and k_j was not requested after time m. The probability for this is

$$\sum_{m=1}^{t} (1-p_i-p_j)^{t-m}\, p_i = \sum_{m=0}^{t-1} (1-p_i-p_j)^{m}\, p_i$$

$$= \left(\frac{p_i}{p_i+p_j}\right) - (1-p_i-p_j)^{t} \left(\frac{p_i}{p_i+p_j}\right)$$

Adding these gives

$$P(k_i \text{ ahead of } k_j \text{ at time } t) =$$

$$\frac{1}{2} (1-p_i-p_j)^{t} + \left(\frac{p_i}{p_i+p_j}\right) - \left(\frac{p_i}{p_i+p_j}\right)(1-p_i-p_j)^{t}$$

$$= \frac{p_i}{p_i+p_j} + \frac{p_i-p_j}{2(p_i+p_j)} (1-p_i-p_j)^{t}$$

Then

$$E(\text{Cost}) = 1 + \sum_{i=1}^{n} p_i \sum_{j\neq i} P(k_j \text{ ahead of } k_i)$$

$$= 1 + \sum_{i=1}^{n} \sum_{j\neq i} \frac{p_i p_j}{p_i+p_j} + \frac{p_i(p_j-p_i)}{2(p_i+p_j)} (1-p_i-p_j)^{t}$$

$$= 1 + 2 \sum_{1\leq i<j\leq n} \frac{p_i p_j}{p_i+p_j} + \sum_{1\leq i<j\leq n} \frac{(p_i-p_j)^2}{2(p_i+p_j)} (1-p_i-p_j)^{t} \qquad \square$$

As $t \to \infty$ the last term vanishes and the first two terms give us the steady state cost. The last term then measures the speed of convergence.

Determining the expected cost of the transposition rule as a function of time is much more difficult and has been determined only in some simple cases. We will consider two cases which will serve to illustrate the difference in the rates of convergence of the two rules.

In the first case, we assume that one key (k_1) has probability one and the other n-1 have probability zero. Using the move to the front rule, the first request will be for k_1, which will immediately move to the front and remain there. The cost is then

$$\frac{n+1}{2} \quad \text{for} \quad t = 0.$$

$$1 \quad \text{for} \quad t > 0.$$

Using the transposition rule, k_1 is equally likely to start in any position and will move up one position at a time until it reaches the top. We will then have

$$\text{Prob}(k_1 \text{ is in position } 1 \text{ at time } t) = \begin{cases} \frac{t+1}{n}, & t \leq n-2 \\ 1, & t \geq n-1 \end{cases}$$

$$\text{Prob}(k_1 \text{ is in position } i \neq 1 \text{ at time } t) = \begin{cases} \frac{1}{n} & \text{if } n-i \leq t \\ 0 & \text{otherwise} \end{cases}$$

For $t \leq n-2$, the expected cost is:

$$1 \cdot \frac{t+1}{n} + \sum_{i=2}^{n-t} i \cdot \left(\frac{1}{n}\right) = 1 + \frac{(n-t)(n-t-1)}{2n} = 1 + \frac{1}{n}\left(\frac{n-t}{2}\right)$$

For $t \geq n-1$ the expected cost is 1 since k_1 must have reached the top.

An interesting statistic to compute from these time varying costs is the overwork. This is defined as the area between the cost

curve and its asymptote. (See Figure 2.2.1) The overwork measures how quickly the cost converges to its asymptote. Also, since the area under a cost curve measures the total cost, the overwork represents the total number of comparisons we do in addition to the asymptotic cost.

The overwork can be determined by summing the time varying part of the equation for the cost. The overwork for the move to the front rule is then

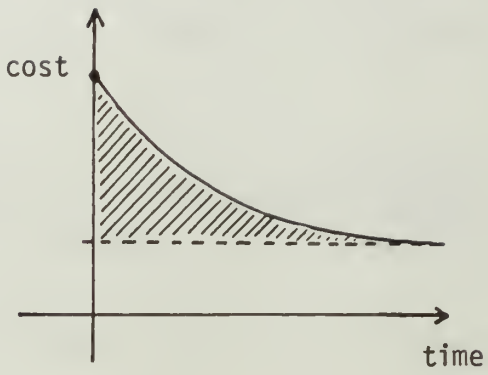
$$\sum_{t=0}^{\infty} \sum_{1\leq i<j\leq n} \frac{(p_i-p_j)^2}{2(p_i+p_j)} (1-p_i-p_j)^t = \sum_{1\leq i<j\leq n} \frac{(p_i-p_j)^2}{2(p_i+p_j)^2}.$$

This formula allows us to get a simple upper bound on the overwork. Since $\left|\frac{p_i-p_j}{p_i+p_j}\right| \leq 1$ we have

$$\sum_{1\leq i<j\leq n} \frac{1}{2} \left(\frac{p_i-p_j}{p_i+p_j}\right)^2 \leq \sum_{1\leq i<j\leq n} \frac{1}{2} = \frac{n(n-1)}{4},$$

so the overwork is $O(n^2)$ for the move to the front rule. This bound is interesting since it tells us how significant the overwork can be compared to the asymptotic cost. For example, after $\frac{n(n-1)}{4}$ key requests, the overwork is <u>at most</u> one comparison per request, and the asymptotic cost is a good approximation to the amount of work we have done.

For the distribution just considered, the overwork in the move to the front rule is $\frac{n-1}{2}$. The overwork in the transposition rule is



The overwork is the area between the cost curve and its asymptote.

Figure 2.2.1 The definition of the overwork.

$$\sum_{t=0}^{n-2} \frac{1}{n} \binom{n-t}{2} = \frac{1}{n} \sum_{t=2}^{n} \binom{t}{2} = \frac{1}{n} \binom{n+1}{3} = \frac{n^2-1}{6}.$$

So we see that the move to the front rule does overwork $\Omega(n)$ and the transposition rule does $\Omega(n^2)$, and hence the move to the front rule approaches its asymptote more quickly. Also note that the move to the front rule converges in 1 request, but the transposition rule requires n-2 requests.

We now consider a slightly more complicated case. Suppose there are n-1 elements of probability $\frac{1}{n-1}$, and one element (k_1) of probability zero. This is not equivalent to the previous case in which k_1 was moved (unless it was at the top) after each request. Now k_1 may or may not move depending on which of the n-1 elements with nonzero probability is accessed.

The overwork for the move to the front rule in this case is $\frac{n-1}{2}$. This can be obtained by substituting the p_i in the overwork formula.

In order to determine the overwork in the case of the transposition rule, we calculate P(k,t), the probability k_1 is in position k at time t. Notice that k_1 will move down only when the key directly under it is accessed and that this occurs with probability $\frac{1}{n-1}$. We then have for $k < n$: $P(k,t) = \sum_{i=1}^{k}$ Prob(k_1 initially in position i)·Prob (k_1 moves down k-i positions in t time steps).

$$= \sum_{i=1}^{k} \frac{1}{n} \cdot \binom{t}{k-i} \left(\frac{1}{n-1}\right)^{k-i} \left(\frac{n-2}{n-1}\right)^{t-k+i}$$

$$= \frac{1}{n} \left(\frac{n-2}{n-1}\right)^{t} \sum_{i=0}^{k-1} \binom{t}{i} \left(\frac{1}{n-2}\right)^{i}$$

For $k = n$ (probability k_1 is in the last position)

$P(n,t) = \sum_{i=1}^{n}$ Prob(k_1 initially in position i) Prob(k_1 moves down n-i positions in $\leq$ t steps).

$$= \sum_{i=1}^{n} \frac{1}{n} \sum_{j=n-i}^{t} \binom{t}{j} \left(\frac{1}{n-1}\right)^{j} \left(\frac{n-2}{n-1}\right)^{t-j}$$

$$= 1 - \left(\frac{n-2}{n-1}\right)^{t} \frac{1}{n} \sum_{j=0}^{n-2} \binom{t}{j} \frac{n-j-1}{(n-2)^{j}}$$

Now if k is the position of k_1, then

$$\text{cost} = \sum_{\substack{j=1 \\ j \neq k}}^{n} \frac{1}{n-1} j = \left(\sum_{j=1}^{n} \frac{j}{n-1}\right) - \frac{k}{n-1} = \frac{(n+1)n}{2(n-1)} - \frac{k}{n-1}$$

Then

$$E(\text{cost}) = \frac{(n+1)n}{2(n-1)} - \frac{E(k)}{n-1} = \frac{(n+1)n}{2(n-1)} - \frac{\sum_{i=1}^{n} iP(i,t)}{n-1}$$

$$= \frac{n}{2} + \frac{1}{n(n-1)} \left(\frac{n-2}{n-1}\right)^{t} \sum_{j=0}^{n-2} \frac{1}{(n-2)^{j}} \binom{t}{j} \binom{n-j}{2}$$

This gives us the expected cost as a function of time. The overwork is then:

$$\sum_{t=0}^{\infty} \frac{1}{n(n-1)} \left(\frac{n-2}{n-1}\right)^t \sum_{n=0}^{n-2} \frac{1}{(n-2)^j} \binom{t}{j} \binom{n-j}{2}$$

$$= \frac{1}{n(n-1)} \sum_{j=0}^{n-2} \frac{1}{(n-2)^j} \binom{n-j}{2} \sum_{t=0}^{\infty} \left(\frac{n-2}{n-1}\right)^t \binom{t}{j}$$

By use of a Taylor Expansion, we can verify

$$\frac{1}{(1-x)^{k+1}} = \sum_{i=0}^{\infty} \binom{k+i}{k} x^i.$$

Using this substitution, we can show the overwork equals $\frac{n^2-1}{6}$, which also is the same as our earlier model. Again, the move to the front rule does $\Omega(n)$ overwork, and the transposition rule does $\Omega(n^2)$ overwork.

For this case, it is also possible to obtain simple bounds on the residual cost, i.e. the difference between the cost and the asymptotic cost. By substituting the p_i into the equation for the cost of the move to the front rule, we get $COST_{MTF} = \frac{n}{2} + \frac{1}{2}\left(\frac{n-1}{n-2}\right)^t$. The residual cost is then $\frac{1}{2}\left(\frac{n-1}{n-2}\right)^t \approx \frac{1}{2} e^{-t/(n-2)}$ for large n.

For the transposition rule, note that if $t \leq n - 2$, all the terms of a binomial expansion are present in the time-varying cost, and the residual cost equals $\frac{1}{2n(n-1)} \frac{t^2-t(2n^2-4n+3)+n(n-1)^3}{(n-1)^2} \approx \frac{t^2-2n^2t+n^4}{2n^4}$ for large n.

If $t > n-2$ we can add terms with $n-2 < j \leq t$ to complete the binomial expansion and obtain this result as an upper bound on the convergence. This bound, however, gets progressively poorer as t becomes larger since we must add more and more terms. In fact, the bound goes to infinity as $t \to \infty$.

These two bounds illustrate the difference rates of convergence. Initially (when $t \leq n-2$) the move to the front rule converges exponentially, and the transposition rule converges quadratically, so the move to the front rule converges considerably more quickly. To give an idea of the magnitude of these bounds, for $t = n-2$, Residual $\text{Cost}_{MTF} \approx .1839$ and at $t = n^2$ Residual $\text{Cost}_{MTF} \approx \frac{1}{2}(e)^{-n^2/n-2} \approx \frac{1}{2}(\frac{1}{e})^n$. On the other hand, Residual $\text{Cost}_{TR} \approx \frac{1}{2}$ at $t = n-2$ and Residual $\text{Cost}_{TR} \leq \frac{1}{2n}$ for $t \approx n^2$.

In general, the transposition rule will converge exponentially, much more slowly than the move to front rule. The convergence of the cost, which is $c_1\lambda_1^t + \ldots + c_n\lambda_n^t$ (see appendix), is mainly determined by the size of the eigenvalues with largest modulus. These are much larger in the case of the transposition rule. As a comparison, for Zipf's Law with 3 elements, the eigenvalues which have nonzero c_i are $(1-p_i-p_j)$ for the move to the front rule (.545, .273, .182). For the transposition rule, these can be numerically calculated as: .710, .576, -.344, .175, -.117. Indeed, the "major" eigenvalues of the transposition rule are larger, and slower convergence will result.

The overwork has been numerically calculated for more complicated distributions. We have already determined a simple form for the

overwork in the move to the front rule and can just put in the particular distribution. For the transposition rule, there is no known simple form. We can closely approximate the overwork by letting $\bar{x}_0 = (\frac{1}{n!},\ldots,\frac{1}{n!})$ be the initial distribution over the states of the Markov chain. Then $\bar{x}_0 P^t$ is the distribution after t requests. From this, we can calculate the expected cost at any t. The asymptotic cost (Acost) can be determined directly from the steady state probabilities, or approximated by the cost of $\bar{x}_0 P^t$ for large t. The overwork is then

$$\sum_{i=0}^{\infty} [\text{cost}(\bar{x}_0 P^i)\text{-Acost}] \approx \sum_{i=0}^{t} [\text{cost}(\bar{x}_0 P^i)\text{-Acost}],$$

for sufficiently large t. This is the quantity we calculate. The overwork for several distributions is shown in Table 2.2.2.

By analyzing the differences between successive values of the overwork in the case of Zipf's Law, we can conclude that the transposition rule does $\Omega(n^3)$ overwork while the move to front rule does only $\Omega(n^2)$. Thus, for a more complicated distribution the transposition rule does much more overwork.

In fact, assuming Zipf's Law, we can derive an exact form for the move to front rule overwork and prove it is $\Omega(n^2)$ and thus the bound of $\frac{n(n-1)}{4}$ is of the right order.

Theorem: Assume that the key request probabilities satisfy Zipf's Law. Then the overwork for the move to front rule with a list of n elements is

Table 2.2.2

The Overwork for Various Distributions

	OVERWORK FOR MOVE TO FRONT RULE
ENGLISH LETTERS	52.7469
ENGLISH WORDS	122.3576

OVERWORK FOR GEOMETRIC DISTRIBUTION WITH N ELEMENTS

N	MOVE TO FRONT RULE
3	0.2911
4	0.8291
5	1.7564
6	3.1250
7	4.9632
8	7.2860
9	10.1011
10	13.4123
11	17.2216
12	21.5299
13	26.3377
14	31.6452
15	37.4527
16	43.7600
17	50.5674
18	57.8747
19	65.6820
20	73.9893

OVERWORK FOR ZIPF'S LAW WITH N ELEMENTS

N	MOVE TO FRONT RULE	TRANSPOSITION RULE
3	0.2006	0.4579
4	0.4463	1.6503
5	0.7978	3.9793
6	1.2576	7.7514
7	1.8272	13.3005
8	2.5076	
9	3.2994	
10	4.2031	
11	5.2189	
12	6.3473	
13	7.5882	
14	8.9420	
15	10.4087	
16	11.9884	
17	13.6812	
18	15.4871	
19	17.4063	
20	19.4387	

$$\frac{5n^2}{12} - (n^2+n+\frac{1}{6})(H_{2n}-H_n) + \frac{1}{6}H_n + \frac{n(n+1)(2n+1)}{3}[H_{2n}^{(2)}-H_n^{(2)}],$$

where

$$H_n^{(2)} = \sum_{i=1}^{n} \frac{1}{i^2}.$$

Asymptotically, this is $(\frac{3}{4} - \ln 2)n^2 \approx .057n^2$.

Proof: Substituting $p_i = \frac{1}{iH_n}$ into the overwork formula gives

$$\frac{1}{2} \sum_{1\le i<j\le n} \frac{(\frac{1}{iH_n} \quad \frac{1}{jH_n})^2}{(\frac{1}{iH_n} + \frac{1}{jH_n})^2} = \frac{1}{2} \sum_{1\le i<j\le n} \frac{(i-j)^2}{(i+j)^2}$$

Since the summand is symmetric in i and j, we have

$$\frac{1}{4}\left[\sum_{1\le i<j\le n} \frac{(i-j)^2}{(i+j)^2} + \sum_{1\le j<i\le n} \frac{(i-j)^2}{(i+j)^2}\right] = \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{(i-j)^2}{(i+j)^2}$$

Now, making the substitution $k = i+j$, we get

$$\frac{1}{4} \sum_{k=2}^{n} \sum_{j=1}^{k-1} \frac{(k-2j)^2}{k^2} + \frac{1}{4} \sum_{k=n+1}^{2n} \sum_{j=k-n}^{n} \frac{(k-2j)^2}{k^2} \qquad (1)$$

The first term in equation (1) equals

$$\frac{1}{4} \sum_{k=2}^{n} \left[\left(\sum_{j=1}^{k-1} 1\right) - \frac{4}{k} \left(\sum_{j=1}^{k-1} j\right) + \frac{4}{k^2} \left(\sum_{j=1}^{k-1} j^2\right) \right]$$

$$= \frac{1}{4} \sum_{k=2}^{n} \left[k-1 - \frac{4}{k} \cdot \frac{(k-1)k}{2} + \frac{4}{k^2} \cdot \frac{(k-1)k(2k-1)}{6} \right]$$

$$= \frac{1}{4} \sum_{k=2}^{n} \left[\frac{k}{3} - 1 + \frac{2}{3k} \right]$$

$$= \frac{1}{4} \left[\frac{n(n+1)}{6} - \frac{1}{3} - (n-1) + \frac{2}{3}(H_n - 1) \right]$$

$$= \frac{n^2}{24} - \frac{5n}{24} + \frac{1}{6} H_n \qquad (2)$$

The second term in equation (1) equals

$$\frac{1}{4} \sum_{k=n+1}^{2n} \left[\left(\sum_{j=k-n}^{n} 1 \right) - \frac{4}{k} \left(\sum_{j=k-n}^{n} j \right) + \frac{4}{k^2} \left(\sum_{j=k-n}^{n} j^2 \right) \right]$$

$$= \frac{1}{4} \sum_{k=n+1}^{2n} \left[(n - (k-n) + 1) - \frac{4}{k} \left(\frac{n(n+1)}{2} - \frac{(k-n-1)(k-n)}{2} \right) \right.$$

$$\left. + \frac{4}{k^2} \left(\frac{n(n+1)(2n+1)}{6} - \frac{(k-n-1)(k-n)(2(k-n)-1)}{6} \right) \right]$$

Collecting all equal powers of k gives

$$\frac{1}{4} \sum_{k=n+1}^{2n} \left[-\frac{k}{3} + (2n+1) - \frac{1}{k}\left(4n^2+4n+\frac{2}{3}\right) + \frac{4}{k^2} \left(\frac{n(n+1)(2n+1)}{3} \right) \right]$$

$$= \frac{1}{4} \left[-\frac{1}{3} \cdot \left(\frac{2n(2n+1)}{2} - \frac{n(n+1)}{2} \right) + (2n+1)n - \left(4n^2+4n+\frac{2}{3}\right)(H_{2n} - H_n) \right.$$

$$\left. + \frac{4n(n+1)(2n+1)}{3} \left(H_{2n}^{(2)} - H_n^{(2)} \right) \right].$$

$$= \frac{3}{8} n^2 + \frac{5}{24} n - (n^2+n+\frac{1}{6})(H_{2n}-H_n)$$

$$+ \frac{n(n+1)(2n+1)}{3} (H_{2n}^{(2)} - H_n^{(2)}).$$

Adding (2) and (3) gives the total result,

$$\frac{5}{12} n^2 - (n^2 + n + \frac{1}{6})(H_{2n}-H_n) + \frac{1}{6} H_n$$

$$+ \frac{n(n+1)(2n+1)}{3} (H_{2n}^{(2)} - H_n^{(2)}). \qquad (3)$$

To determine the asymptotic behavior, note that $H_n \approx \ln n$, so $H_{2n} - H_n \approx \ln 2n - \ln n = \ln 2$, so the second term is asymptotically $n^2 \ln 2$. The third term is $O(\log n)$ and is dominated by the n^2 terms.

Finally, we need to approximate $H_{2n}^{(2)} - H_n^{(2)} = \sum_{i=n+1}^{2n} \frac{1}{i^2}$. Since the summand is a decreasing function, we can bound it using the following relation:

$$\int_a^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq f(a) + \int_a^b f(x)dx$$

Substituting $a = n+1$, $b = 2n$ and $f(x) = \frac{1}{x^2}$ gives

$$\int_{n+1}^{2n+1} \frac{dx}{x^2} \leq \sum_{i=n+1}^{2n} \frac{1}{i^2} \leq \frac{1}{(n+1)^2} + \int_{n+1}^{2n} \frac{dx}{x^2}$$

$$\frac{n}{(2n+1)(n+1)} \leq \sum_{i=n+1} \frac{1}{i^2} \leq \frac{n^2 + 6n - 1}{2n(n+1)^2}$$

Since both the upper and lower bounds equal $\frac{1}{2n} + O(n^{-2})$, we have

$\sum_{i=n+1}^{2n} \frac{1}{i^2} = \frac{1}{2n} + O(n^{-2})$ and the fourth term in (4) approaches $\frac{1}{3} n^2$. Hence, the asymptotic value for (4) is

$$\frac{5}{12} n^2 - n^2 \ln 2 + \frac{1}{3} n^2 = (\frac{3}{4} - \ln 2)n^2 \approx .057n^2$$ □

We can get a graphic idea of the difference in convergence from Figure 2.6.1 and Table 2.6.1 in Section 2.6. These show the cost of accessing a list ordered by the two rules as a function of time and compare them with the frequency count rule (see Section 2.6) which is optimal. From graphs like these, it is interesting to calculate the smallest number of requests for which it is better to use the transposition rule (See Table 2.2.3). Note that the value we are really interested in is <u>not</u> the point where the two cost curves cross, but the point where the <u>integrals</u> of the two curves cross. This is because we want the rule that does the least <u>total</u> work.

The slope of the cost crossover in Table 2.2.3 increases, so it is super linear and may be about $\Omega(n \log n)$. The integral crossover appears to be $\Omega(n^2)$. We can also get an estimate of the integral crossover as follows: If we assume all the overwork has been done by time t, the integral crossover time, then the cost integral for the move to front rule is t times the asymptotic cost (AS_m) plus the overwork (OV_m), and similarly for the transposition rule. Since we are at the point where these integrals cross,

$$t \cdot AS_m + OV_m = t \cdot AS_{TR} + OV_{TR}$$

$$t = \frac{OV_{TR} - OV_m}{AS_m - AS_{TR}} .$$

Table 2.2.3

Cost Crossover and Integral Crossover Times

n	Cost Crossover	Integral Crossover
3	3	6
4	5	10
5	7	14
6	10	20
7	13	27
10	22	50
20	75	212

Points where the cost and integral of the cost for the transposition rule become less than that of the move to front rule, for an n-element list with Zipf's Law as the probability distribution.

Earlier in this section, we found $OV_{TR} = \Omega(n^3)$ and $OV_m = \Omega(n^2)$. Since the asymptotic costs are bounded within twice the optimal cost (which is $\frac{n}{\ln n}$), $AS_m - AS_{TR} = \Omega(\frac{n}{\ln n})$ and hence we get $t = \Omega(n^2 \ln n)$, which is slightly larger than shown in Table 2.2.3.

In summary, though the transposition rule has lower asymptotic cost than the move to front rule, it converges to that cost much more slowly, and, in fact, for Zipf's law, it will require $\Omega(n^2)$ key requests before it becomes more economical to use the transposition rule.

2.3 Other Permutation Rules

We previously defined the idea of a permutation rule, where a permutation τ_i is perfomed on the list when the key in location i is requested. So far, we have only considered two such rules: the move to front rule and the transposition rule. There are a total of $(n!)^n$ possible such rules, but most will just senselessly jumble the list, resulting in no decrease in cost.

Let us think intuitively about what a "sensible" rule must be like. We will see that a sensible rule should move the requested key up in the list by a certain amount (which may depend on the location of the requested key). This is the only good way to use the information that this key, having been requested, should have higher probability. Any permutation not of this form can be viewed as performing first a sensible permutation and then a permutation that leaves the requested element alone. This second permutation will only increase the disorder of the list since no additional information has been given on these keys, and permuting them will work against the order we are trying to create.

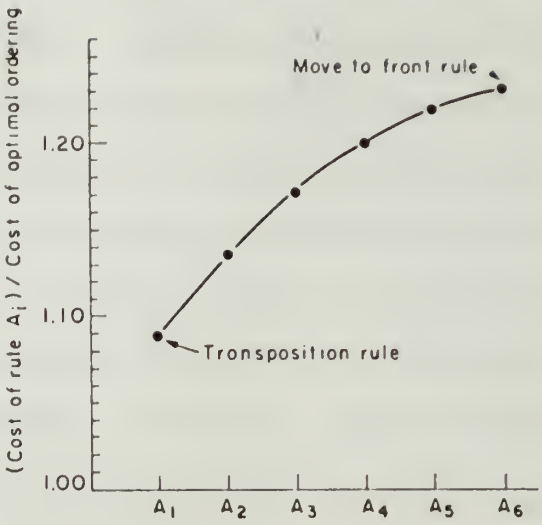
We consider the following sort of sensible rule which moves the requested key k position ahead for some fixed k. Another type of rule that should behave similarly is where the requested key is moved some fixed fraction of the distance to the top.

It can be seen from Figure 2.3.1 (due to Rivest [2]) and Table 2.3.1 that as the distance the requested key moves is increased, the asymptotic cost increases and the rules converge more quickly, forming a spectrum of rules, ranging from the move ahead 1 (transposition) rule at one end to the move ahead n-1 (move to front) rule at the other.

2.4 A Hybrid Rule

We can get a rule that is superior to any of those we have considered so far by relaxing the restraint that the rule cannot vary with respect to time. A hybrid rule can be envisioned that moves keys to the front for some initial period of time, then switches and begins transposing. Such a rule will enjoy the advantages of both rules. Initially, it will move keys to the front and will therefore converge quite rapidly. Asymptotically, it will behave like the transposition rule and therefore will have a low asymptotic cost.

The question is when we should switch rules. To help answer this question, a simulation was run using Zipf's Law for the key request probabilities. Each trial of the simulation used the move to front rule until the expected decrease from using the transposition rule became larger than that of the move to front rule. The number of requests required for this to occur is an approximation to the correct



This figure, due to Rivest [2], compares the cost of different move ahead k rules (A_i refers to the move ahead i rule) for a list of seven elements whose probabilities are given by Zipf's Law.

Figure 2.3.1 Asymptotic comparison of the move ahead k rules.

Table 2.3.1

Comparison of the Convergence of the Move Ahead k Rules

k	Lowest Cost	Lowest Total Cost
5	0 - 3	0 - 5
4	4	6 - 8
3	5 - 7	9 -13
2	8 -15	14 -38
1	16 - ∞	39 - ∞

The results of a simulation using a list of 6 elements whose probabilities are given by Zipf's Law show the time interval for which each move ahead k rule has lowest cost and lowest total cost (the total cost is the cost summed over all previous requests).

time to switch. These times were then averaged over all trials to give the results shown in Table 2.4.1. The results of this simulation indicate .268n + .980 as the best time to switch rules. This time, of course, depends on the request probabilities, but we would not expect it to vary too much for different distributions. Furthermore the choice is not too critical. Since the transposition rule converges so slowly, little is lost if we use the move to front rule for too long. We need only make sure that our choice is large enough to have the move front rule be close to its asymptote. We would then switch after .5n requests, to make sure we had used the move to front rule long enough to significantly reduce the cost.

Another method would be to estimate our position on the cost curve by counting the number of compares we require and averaging over a period of time. Once this estimate stops decreasing, we suspect that we are in the flat part of cost curve, and we switch to the transposition rule. This rule has the overhead of counting the number of comparisons. In addition, we must be careful not to average over too short a period, or we may switch too soon.

This rule is best employed when we expect an intermediate number of requests. If few ($O(n^2)$) requests are expected, then the move to front rule is used. A great number suggests the transposition rule. An intermediate number means that <u>both</u> of the good features of the hybrid (fast convergence and low cost) will be valuable and the overhead incurred by using this rule will be worthwhile.

Table 2.4.1

Best Times to Switch Rules

n	Average Switch Time
3	1.90
4	2.20
5	2.29
6	2.52
7	2.84
8	2.94
9	3.35
10	3.55
20	6.608
30	8.924

Simulation showing the average best time to switch from the move to front rule to the transposition rule. The probability distribution is Zipf's Law over n elements.

2.5 The First Request Rule

The first request rule is defined as follows: the first time a key is requested, it is moved up in the list until it comes to the top or a previously requested key. After that, it is not moved. Note that the keys occur in the list in order of their first request. After all keys have been requested, the ordering obtained is the same as if the keys had not been known a priori, and the list had been built by inserting a "new" key (one that had been requested for the first time) at the end of the list.

The following theorem characterizes the performance of this rule.

Theorem: Given any initial list, the probability of obtaining a given final list after any number of requests is the same for the move to front and first request rules.

Proof: Consider any sequence of requests $r_1 \ldots r_k$ as inputs to the move to front rule, and the reverse sequence $r_k \ldots r_1$ as inputs to the first request rule. Note that these two sequences have the same probability. Suppose that both rules start with the same list. We now show that these two sequences produce the same ordering. Consider any two keys k_i and k_j. If neither is requested, both rules will leave the initial order unchanged, and k_i and k_j will be ordered the same in the two final lists. If only one (say k_i) is requested, then both rules will have k_i ahead of k_j in the final list. If both are requested (say k_i is requested after k_j in the sequence $r_1 \ldots r_k$), then k_i will be ahead

of k_j in the move to front list. Since k_i is requested before k_j in the sequence $r_k \ldots r_1$, k_i will also be ahead of k_j in the first request list, hence the orderings in the final list will again be the same. In any case, k_i is ahead of k_j in one list if and only if it is ahead of k_j in the other. Hence the two lists must have the same ordering.

Now consider any list. For each sequence of requests that will produce this list using one rule, there exists a sequence of equal probability that will produce this same list using the other rule. Hence the probability for either rule to produce this list must be equal. □

This theorem is easily extended to hold for a probability distribution over initial lists since the two rules will behave identically for each initial list. Also, it implies that the cost of the first request rule at any time will equal the cost of the move to front rule. Therefore all the previous results concerning the move to front rule apply to the first request rule.

Suppose the keys were not known a priori and the list was constructed by inserting a "new" key at the end of the list. Clearly, the asymptotic distribution will be that of the first request rule. This theorem tells us that if the initial list was constructed in this manner, using the move to the front rule will not decrease the cost (since the Markov chain will be in steady state).

The first request rule differs from the move to front rule in two important ways. First, since each key is moved only once, it is

cheaper to execute than the move to front rule. Second, since the list converges to a specific ordering (which may have very high cost), the variance of the cost is much higher than that of the move to front rule.

The first request rule can be modeled by a Markov chain with $(n+1)n!$ states. For each of the $n!$ orderings of the list, the chain can be in $n+1$ different states, depending on whether $0,1,\ldots,$ or n different keys have been requested. Unlike previous chains, this chain is reducible (see appendix). Once we reach a state in which all n keys have been requested, we are "trapped" and cannot leave this state.

On the other hand, an irreducible chain cannot get trapped and must divide its time among all states that have nonzero steady state probability. In fact, the ergodic theorem tells us that if a state has steady state probability p, the chain will spend a fraction of its time equal to p in this state.

We are now in a position to talk about the variance of the costs of these two rules. If we let c_i be a random variable equal to the cost of the state the chain is in at time i, $E(c_i)$, $VAR(c_i)$ and $E(\frac{c_1+c_2+\ldots+c_i}{i})$ are the same for both rules. However the variance of the cost averaged over some time period $[VAR(\frac{c_1+\ldots+c_n}{n})]$ is much greater for the first request rule. The fact that the move to front rule is irreducible implies $\lim_{n\to\infty} VAR(\frac{c_1+\ldots+c_n}{n}) = 0$ (see appendix). However, for large n, $c_n = c_{n+1}$ using the first request rule (since the chain has reached a final state), and $\frac{c_1+\ldots+c_n}{n} \approx c_n$. Therefore the variance

of the average cost is $VAR(c_n) > 0$. (An expression for this variance can be found in McCabe [10].)

We can use the first request rule to form a hybrid rule with the transposition rule as follows: When a key is first requested, we use the first request rule and move the key up until a previously requested key is encountered. When the key is subsequently requested, we use the transposition rule to promote it in the list.

The performance of this hybrid is better then the move to front/transpose hybrid. The only requests handled by the transposition rule are second and subsequent requests. Hence the initial list that the transposition part of the hybrid "sees" is a list ordered by the first request rule, which, as we have seen, is the move to front rule (or first request rule) steady state. Hence the transposition rule "starts" from the move to front rule steady state. This is an improvement over using the move to front rule initially since then the steady state is never reached. In addition, this hybrid will reduce the cost more quickly than the first request rule because it does a cost reducing transposition on second and subsequent requests of a key, while the first request rule alone does nothing. This hybrid also has the desirable feature that no guesswork need be done as to when to switch rules. This choice is performed automatically by the algorithm.

2.6 Frequency Count Rule

Perhaps the most natural way to cause high frequency keys to move to higher positions in the data structure would be to keep count

of how many times each key has been requested. If we assume the request probabilities are constant with respect to time and then keep the keys sorted according to their frequency counts, high probability keys will move to the top.

The primary advantage of this rule is that it has a lower access time than the other rules we have considered. In fact, its performance is optimal. In addition, frequency information is available for analysis, which may be desirable, and the changes required to execute the rule are quite simple. The primary disadvantage is that count fields must be kept, requiring extra storage. These points are now considered in greater detail.

We first discuss the performance of this rule. The following theorem shows that it is asymptotically optimal.

Theorem: As the number of requests, $t \to \infty$, a list ordered by the frequency count rule approaches the optimal ordering.

Proof: If two keys k_i and k_j have probabilities p_i and p_j with $p_i > p_j$, the probability that k_i is ahead of k_j after t requests approaches one as $t \to \infty$. Since $E(\text{Cost}) = \sum_{i=1}^{n} p_i(1 + \sum_{j \neq i} \text{Prob}(k_j \text{ is ahead of } k_i))$ we have $\lim_{t \to \infty} E(\text{Cost}) = \sum_{i=1}^{n} ip_i$ which is the optimal cost. □

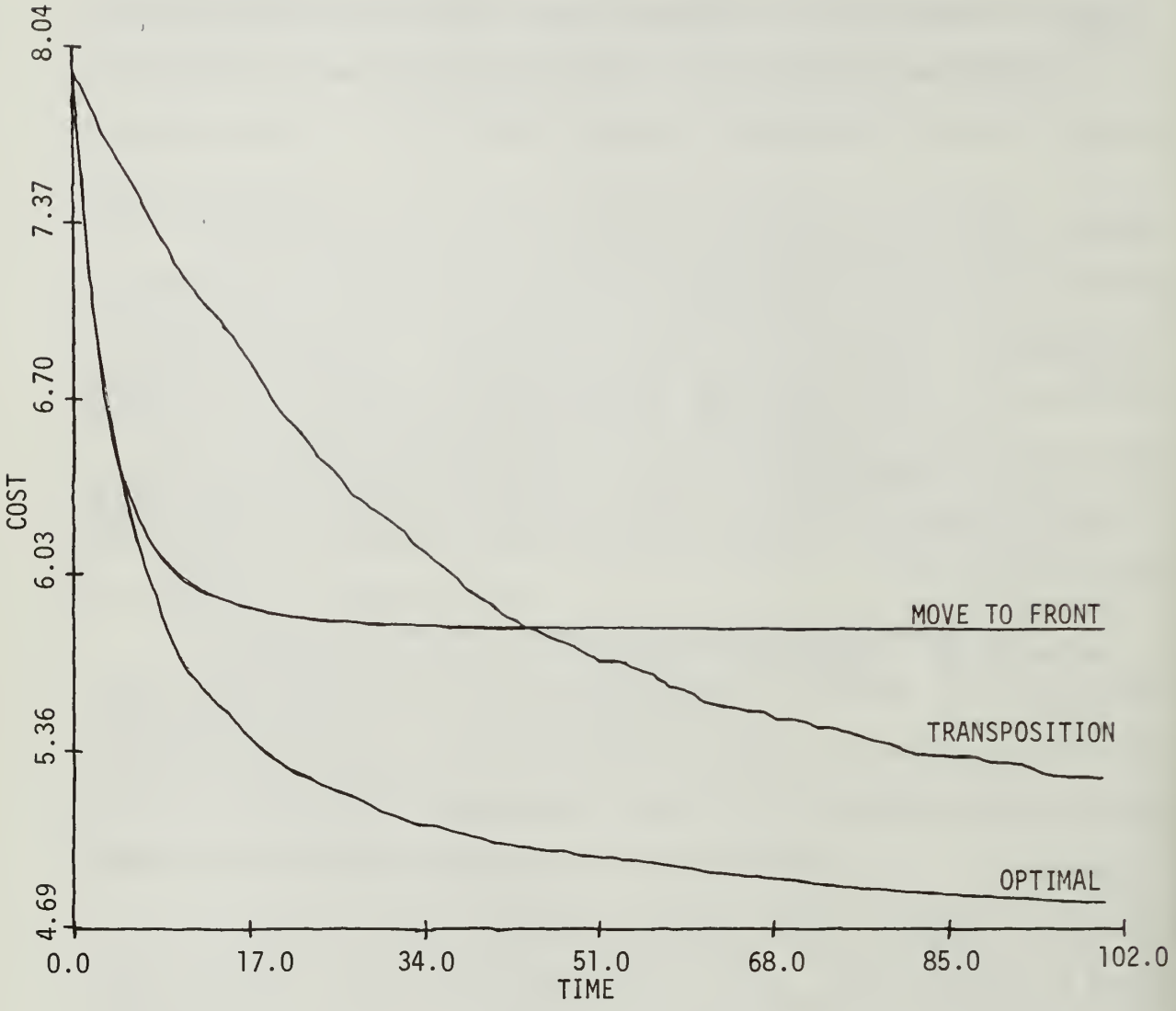
Also, if we have no a priori reason to suspect k_i is more probable than k_j, this rule is optimal at any time.

Theorem: If we have no a priori knowledge of the probability distribution, the frequency count rule provides the optimal ordering at any time.

Proof: If we have no a priori knowledge, then all distributions of key requests must be considered equally likely, so if k_i has occurred more times than k_j, $\text{Prob}(p_i > p_j) > \text{Prob}(p_i < p_j)$, and an arrangement with k_i ahead of k_j will have a lower expected cost. Clearly, the arrangement with the lowest expected cost will be the one in which the keys are sorted by frequency count, and this, of course, is the arrangement given by the count rule. □

A comparison with previous rules is given by Figure 2.6.1, which shows the results of a simulation done on a 15-element list using Zipf's Law. Table 2.6.1 shows a simulation for a list with 100 elements. These two simulations give us a good idea of the differing rates of convergence of the two previous rules and how they compare to the optimum. Initially, the move to the front cost decreases nearly as quickly as that of the count rule. This is intuitively reasonable: Initially, the count rule will move the requested item close to the top, so its behavior should be very close to the move to the front rule. On the other hand, the transposition rule's cost decreases very slowly, especially on the 100 element list.

As mentioned before, the changes required by this rule after each request are small. Suppose k_i is requested for the r^{th} time. The only change is to increase k_i's frequency count from r-1 to r and move



Simulation on a 15-element list using Zipf's Law
"Time" is measured as the number of requests

Figure 2.6.1 Comparison of Various Rules

Table 2.6.1

Another Comparison

TIME	TIME VARYING COST FOR ZIPF'S LAW WITH 100 ELEMENTS MOVE TO FRONT	TRANSPOSITION	FREQUENCY COUNT
0	50.1322	50.1322	50.1322
1	47.4562	50.0749	47.4556
2	45.4688	50.0307	45.4608
3	43.4281	49.9742	43.4257
4	41.8796	49.9329	41.8443
5	40.7099	49.8780	40.6515
6	39.7503	49.8341	39.6460
7	38.5905	49.7742	38.4920
8	37.8538	49.7196	37.6994
9	36.5972	49.6666	36.4395
10	36.1294	49.6216	35.8960

it ahead of all keys having frequency count r-1. We can easily determine to where k_i should move in the following manner. During our search for k_i we keep a pointer to the key furthest down in the list whose count is greater than the key we are currently examining. When we examine k_i, this pointer will point to k_i's new location. Note that after many requests, the count fields will be widely separated, and these moves will rarely be required.

The primary disadvantage of this rule is the additional storage required for the count fields. The storage required, however, can be reduced using very simple techniques. From the updating algorithm, we can see that actual count a key has is not important. What matters is the difference between successive counts, because this gives us all the information we need to keep the keys ordered with respect to count. If we store this difference instead of the full count, we will require

less storage, since the rate of growth of the difference fields is proportional to the difference in successive probabilities (which is small), while the count fields grow in proportion to the probabilities. Note that only a small amount of work is required to update the difference fields since after a request, only once count field changes, and hence at most two difference fields must be updated.

Thus, the count rule is a very attractive rule. Asymptotically it approaches the optimal ordering. At any time, it provides us with the list which has lowest cost, based on the requests we have seen so far. The work required to update the list is also very small. The primary disadvantage is the extra storage required. However, this disadvantage can be reduced by storing the differences between successive counts.

2.7 Limited Difference Rules

We now consider a set of rules which limit the size of the difference fields in the frequency count rule. Once a difference field reaches this limit, additional requests of the more frequent key leave this field unchanged (requests to the other key, of course, decrease this field).

If the maximum difference is zero, then the algorithm will move a key to the front when it is requested, and will perform exactly like the move to front rule. As the maximum difference is increased, the performance will improve, with the full count rule (no maximum difference) as the limit. Therefore, performance approaches the optimum as the number of bits is increased.

To see how much the performance is effected by the number of bits, let us consider a list with only 2 elements, having probabilities a and b(=1-a). If the maximum difference is at most n, then the corresponding Markov chain has 2n+2 states:

A_i, $0 \leq i \leq n$ where the key with probability a is first in the list and the difference is i.

B_i, $0 \leq i \leq n$ where the key with probability b is first in the list with difference i.

It is easy to verify that the steady state equations are:

$$A_n = aA_{n-1} + aA_n \qquad B_n = bB_{n-1} + bB_n$$

$$A_i = aA_{i-1} + bA_{i+1} \qquad B_i = bB_{i-1} + aB_{i+1}, \quad 2 \leq i \leq n-1$$

$$A_1 = bA_2 + aA_0 + aB_0 \qquad B_1 = aB_2 + bB_0 + bA_0$$

$$A_0 = bA_1 \qquad B_0 = aB_1$$

and, in addition, $\sum_{i=0}^{n} A_i + \sum_{i=0}^{n} B_i = 1$.

We solve this system of equations to get

$$A_0 = bA_n\left(\frac{b}{a}\right)^{n-1} \qquad B_0 = aA_n\left(\frac{b}{a}\right)^{n+1}$$

$$A_i = A_n\left(\frac{b}{a}\right)^{n-i} \qquad B_i = A_n\left(\frac{b}{a}\right)^{n+i} \qquad 1 \leq i \leq n$$

and $A_n = \dfrac{a-b}{a\left[1 - \left(\frac{b}{a}\right)^{2n+1}\right]}$

The cost of the list is

(a+2b) Prob(key with probability a is first in list) +

(b+2a) Prob(key with probability b is first)

$$= (a+2b) \sum_{i=0}^{n} A_i + (b+2a) \sum_{i=0}^{n} B_i$$

$$= (1+a) + \frac{2b(b-a)(\frac{b}{a})^n - (b-a)}{[(\frac{b}{a})^{2n+1} - 1]}$$

Let us now suppose that $b > a$. Then the optimal cost is $b+2a = b+a+a = 1+a$, which is the first term in the cost expression. The difference from the optimum is then given by

$$\frac{2b(b-a)(\frac{b}{a})^n - (b-a)}{[(\frac{b}{a})^{2n+1} - 1]} \approx \frac{2b(b-a)}{(\frac{b}{a})^{n+1}} \quad \text{since } \frac{b}{a} > 1.$$

Hence we see that the "use" of adding one to the maximum difference decreases expontially with base $\frac{b}{a}$. This tells us that the performance should be improved by the addition of just a few bits. However, the "flatness" of the distribution (determined by how close $\frac{b}{a}$ is to one in this simple case) determines how many bits will be required. The flatter the distribution, the more bits will be required to correctly distinguish the more probable elements.

Table 2.9.1 shows the results of a simulation run on larger bits. Even using a small maximum difference provides nearly optimal results.

The limited difference rule lets us use a limited amount of storage, while providing nearly optimal results. For a two element list, the cost of this rule approaches the optimum exponentially as we increase the maximum difference.

2.8 Wait c, Move and Clear Rules

We now consider two classes of rules that use bit fields to store information about key requests. The first class uses the bit field as a counter, initially zero, that is incremented by one each time the key is accessed. Once the field exceeds to maximum value, the key is moved (using either the move to front or transposition rule) and the field of every key is reset to zero. The cost of performing this may be very significant. However, if all fields are stored in one area (instead of being directly associated with each key) we can set all fields to zero by zeroing a contiguous area of core, which may be done very efficiently. We will call these rules "wait c, move and clear" rules, where c is the maximum value of the field.

A second class of rules (discussed in the next section) behaves in a similar fashion, except that when a key is moved, only its field is reset to zero. These rules will be called "wait n and move" rules.

In analyzing these rules, we will find that using the count fields in the first manner will decrease the asymptotic cost more than the second method. However, the convergence of the first method will be much slower, since we will not move a key every request, and, if the maximum difference is very large, we will move keys only very rarely.

We begin our analysis of the wait c, move and clear rules with the following theorem.

Theorem: Given key request probabilities $p_1, p_2, \ldots p_n$, the steady state probability of a given list using a wait c, move and clear rule is equal to the steady state probability of the list using the corresponding permutation rule with modified key request probabilities

$$\hat{p}_1(c),\ \hat{p}_2(c),\ldots,\hat{p}_n(c),$$

where

$$\hat{p}_i(c) = \sum_{a_1=0}^{c} \cdots \sum_{a_{i-1}=0}^{c} \sum_{a_{i+1}=0}^{c} \cdots \sum_{a_n=0}^{c}$$

$$\frac{(c+a_1+\ldots+a_{i-1}+a_{i+1}+\ldots+a_n)!}{c!a_1!\ldots a_{i-1}!a_{i+1}!\ldots a_n!} \times$$

$$p_i^{c+1} p_1^{a_1} \ldots p_{i-1}^{a_{i-1}} p_{i+1}^{a_{i+1}} \ldots p_n^{a_n}.$$

Proof: Consider the sequence of keys that have been moved by the wait c, move and clear rule. We have assumed that any two requests are independent, and that the request probabilities are constant with respect to time. Because of these assumptions and the fact that we clear the counts after each move, the move sequence has the following properties:

(1) Any two moves are independent.

(2) The probability that the i^{th} move is a given key does not depend on i.

If we use the move sequence as inputs to a permutation rule, the resulting list will be the same as one obtained by inputting the original request sequence to the wait c, move and clear rule. We note that the properties of the move sequence are exactly those required for a request sequence, so the inputs to the permutation rule can be thought of as a sequence of requests. However, elements of this sequence are not chosen using the request probabilities, but using the probability that a key is moved. The probability that p_i is moved is exactly the $\hat{p}_i$ shown in the statement of this theorem.

This formula is derived as follows: If k_i was moved, we know that k_i has been requested c+1 times, and that the last request (the one that caused k_i to be moved) must have been for k_i. Then for $j \neq i$, let k_j be requested a_j times ($0 \leq a_j \leq c$) and sum over all possible choices for the a_j.

This would complete the proof if every request to the wait c, move and clear rule caused a move. This is not the case since we must wait after each move while the counts build up. If this waiting time were dependent on the current state (as it will be for the wait c and move rules), states with longer waiting times would have proportionally greater probabilities. Fortunately, this is not the case. After each move, the counts are reset and hence each state will have the same expected waiting time. □

This proof demonstrates the reason wait c, move and clear rules outperform permutation rules. In order to be moved, a low probability

key must be requested c+1 times before any other key is requested c+1 times. Hence these are less likely to be moved. On the other hand, high probability keys now have a proportionally greater chance. Notice, of course, that the probability that a key is requested remains the same; we are only being more selective about which key we move.

Due to this correspondence between wait c, move and clear rules and permutation rules, many results from previous sections carry over. Specifically:

Corollary: Let keys $k_1, k_2, \ldots, k_n$ have request probabilities $p_1, p_2, \ldots p_n$ and let $\hat{p}_1(c), \ldots, \hat{p}_n(c)$ be defined as in the previous theorem. Then

(1) The asymptotic cost of the wait c, move to front and clear rule is

$$1 + \sum_{i \neq j} \frac{p_i \hat{p}_j(c)}{\hat{p}_i(c)\hat{p}_j(c)}$$

(2) For the wait c, transpose and clear rule, the steady state probability of any given ordering $(k_1 \ldots k_n)$ is

$$\frac{\prod_{i=1}^{n} \hat{p}_i(c)^{n-i}}{N},$$

where N is a normalizing constant.

(3) The wait c, transpose and clear rule has asymptotic cost less than or equal to that of the wait c, move to front and clear rule.

Proof: All result from replacing p_i(the probability a key is moved by a permutation rule) by $\hat{p}_i(c)$(the probability for a wait c, move and clear rule). □

As in the case of the limited difference rule, the performance approaches the optimum as $c \to \infty$.

Theorem: As $c \to \infty$, the asymptotic costs of the wait c, move to front and clear rule and the wait c, transpose and clear rule approach the optimal cost.

Proof: We first examine the wait c, move to front and clear rule. Consider the probability that k_i is ahead of k_j in the list. This will be the case if any only if k_i was moved at the most recent time when either k_i or k_j was moved (i.e. k_i was the most recently moved of k_i and k_j). Thus, the probability is Prob(k_i was moved | k_i or k_j was moved). This equals the probability that k_i was requested c+1 times before k_j was requested c+1 times. By the law of large numbers, this approaches 1 if $p_i > p_j$ and 0 if $p_i < p_j$. Hence the expected cost which equals

$$1 + \sum_{i \neq j} p_i \text{ Prob}(k_j \text{ ahead of } k_i)$$

approaches

$$1 + \sum_i p_i(i-1) = \sum_i ip_i,$$

the optimal cost.

By (3) of the previous corollary, the wait c, transpose and clear rule has cost less than or equal to that of the wait c, move to front and clear rule, so it also approaches the optimum. □

So both the wait c, move and clear rules and the limited difference rule approach the optimum as the number of bits they use increases. The important question is: which converges more quickly? Table 2.9.1 shows the limited difference rule makes "better use" of its bits.

This can also be demonstrated in the case of a list of two elements (A and B) having probabilities a and b (=1-a). Here the probability that A is ahead of B equals $\sum_{i=0}^{c} \binom{c+i}{i} a^{c+1} b^{i}$. Table 2.8.1 shows this probability approaches one much more slowly than that of the limited difference rule.

A major disadvantage of the wait c, move and clear rules is that they decrease the cost more slowly than the corresponding permutation rule with modified probabilities, since a counter must exceed c for a move to be done. The worst case occurs when every key is requested c times before any key is requested c+1 times. In this case, a move will be done every cn+1 requests. Thus, the convergence can be slowed by a factor $\Omega(n)$. On the other hand, the best case occurs when the same key is requested c+1 times. Here, a move will be made every c+1 requests and the convergence must be slowed by at least this constant multiple.

Table 2.8.1

Probability A is ahead of B for a=.6

C	LIMITED BIT RULE	WAIT C RULE
0	0.60000	0.60000
1	0.66316	0.64800
2	0.74218	0.68256
3	0.81039	0.71021
4	0.86446	0.73343
5	0.90511	0.75350
6	0.93457	0.77116
7	0.95536	0.78690
8	0.96977	0.80106
9	0.97963	0.81391
10	0.98632	0.82562
11	0.99084	0.83636
12	0.99387	0.84623
13	0.99591	0.85535
14	0.99727	0.86379
15	0.99818	0.87162
16	0.99878	0.87890
17	0.99919	0.88569
18	0.99946	0.89202
19	0.99964	0.89794
20	0.99976	0.90348
21	0.99984	0.90868
22	0.99989	0.91355
23	0.99993	0.91812
24	0.99995	0.92242
25	0.99997	0.92647
26	0.99998	0.93028
27	0.99999	0.93387
28	0.99999	0.93725
29	0.99999	0.94045
30	1.00000	0.94346
31	1.00000	0.94631
32	1.00000	0.94900
33	1.00000	0.95154
34	1.00000	0.95395
35	1.00000	0.95623
36	1.00000	0.95838
37	1.00000	0.96042
38	1.00000	0.96236
39	1.00000	0.96419
40	1.00000	0.96593
41	1.00000	0.96757
42	1.00000	0.96914
43	1.00000	0.97062
44	1.00000	0.97203
45	1.00000	0.97336
46	1.00000	0.97463
47	1.00000	0.97584
48	1.00000	0.97698
49	1.00000	0.97807
50	1.00000	0.97910

To get an idea of the average decrease in convergence, we consider n equally likely keys and c=1. Note that this is the least favorable key distribution. We now determine the expected number of requests before a key is requested for a second time. This is

$$\sum_{i=0}^{n} \text{Prob(no key has been requested twice after i requests).}$$

This probability equals the number of sequences of length i of distinct keys ($\frac{n!}{(n-i)!}$) divided by the total number of sequences (n^i).

$$= \sum_{i=0}^{n} \frac{n!}{(n-i)!} \frac{1}{n^i}$$

Replacing i by n-i gives

$$n! \sum_{i=0}^{n} \frac{n^{-(n-i)}}{n!} = n!n^{-n} \sum_{i=0}^{n} \frac{n^i}{i!} < n!n^{-n}e^n$$

Stirling's approximation gives

$$\approx (n^n e^{-n} \sqrt{2\pi n})\, n^{-n}e^n = \sqrt{2\pi n}.$$

Therefore, for c=1, the expected slowdown is $\Omega(\sqrt{n})$, for this unfavorable key distribution.

The wait c, move and clear rules have an interesting correspondence with the permutation rules. They perform better than permutation rules because they are more selective about which keys are moved. However, the performance is not as good as the limited difference rule.

These rules have the further disadvantage of converging more slowly than permutation rules. For a list of n elements, the convergence is slowed by a factor between c+1 and nc+1 times. If c=1, the average slowdown is $\approx\sqrt{2\pi n}$ for the uniform distribution.

2.9 Wait c and Move Rules

We now turn our attention to the wait c and move rules, and first consider the wait c and move to front rule.

Theorem: Given key probabilities $p_1, p_2, \ldots, p_n$, the asymptotic cost of the wait c and move to front rule is

$$1 + \sum_{i \neq j} p_i \, X_{ji},$$

where

$$X_{ji} = \frac{p_j}{(p_i+p_j)(c+1)^2} \sum_{k=0}^{c} (c-k+1)\left(\frac{p_i}{p_1+p_j}\right)^k \sum_{m=0}^{c} \binom{m+k}{m}\left(\frac{p_j}{p_i+p_j}\right)^m$$

(the probability k_j is ahead of k_i in the list).

Proof: Recall that the expected cost is

$$1 + \sum_{i \neq j} p_i \, \text{Prob}(k_j \text{ ahead of } k_i)$$

and therefore we must determine this probability. Consider any two keys, A and B, having probabilities a and b. Note that the relative ordering of A and B will not be effected when another key is moved. Also, their counts will remain the same since they are not cleared. Therefore, in determining Prob(A ahead of B), we can ignore all other keys and requests

to all other keys; we need only consider a list consisting of A and B, having probabilities $\frac{a}{a+b}$ and $\frac{b}{a+b}$ of being requested. (For simplicity, we rename these probabilities "a" and "b").

This list can be modeled by a Markov chain with $2(c+1)^2$ states, A_{ij} and B_{ij} for $0 \leq i, j \leq c$. State A_{ij} corresponds to the list with A (having count i) ahead of B (having count j). State B_{ij} corresponds to B (having count j) ahead of A. Note that the first subscript is always A's count.

Before solving for the stationary distribution, we must first make sure it will give us Prob(A ahead of B). There are two possible troubles. First, as with the wait c, move and clear rule, we must wait in each state of the two element chain while keys other than A and B are being requested. However, since key requests do not depend on whether A is ahead of B, or the count of either key, the requests are independent of the state and hence the expected waiting time is the same for each state.

Second, the chain is periodic with period c+1. If we let r_A and r_B be the number of times A and B have been requested, we have $i \equiv r_A \bmod(c+1)$ and $j \equiv r_B \bmod(c+1)$. Therefore $i+j \equiv (r_A + r_B) \bmod(c+1)$. Since each transition increases $r_A + r_B$ by one, if we start at A_{ij} (or B_{ij}), it will always take a multiple of (c+1) transition to return. Hence the chain has period c+1.

A chain which is periodic does not converge to its steady state distribution in the sense that $\lim_{t\to\infty} P_t(x_0,x) = p(x)$, where $P_t(x_0,x)$

is the probability of going from an initial state x_0 to state x in t transitions, and p(x) is the steady state probability of state x. However, for an irreducible chain (which this one is), the ergodic theorem holds (see appendix). This states that $\lim_{t\to\infty} \frac{1}{t} \sum_{i=0}^{t} P_t(x_0,x) = p(x)$. Hence the "time average" of the probability approaches the steady state distribution. If C(t) is the expected cost at time t, we are guaranteed $\lim_{t\to\infty} \frac{1}{t} \sum_{i=0}^{t} C(i) = \sum_{x} p(x)c(x)$ where c(x) is the cost of state x. The cost converges to the asymptotic cost in this sense. Note that the asymptotic cost is still the stationary probability of a state times its cost summed over all states, only the strength of convergence has been changed.

We now proceed to determine the stationary probability. The steady state equations are:

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1}$$

$$B_{ij} = aB_{i-1,j} + bB_{i,j-1} \qquad \text{for } 0 < i,j \leq c$$

$$A_{0j} = bA_{0,j-1} + aA_{cj} + aB_{cj}$$

$$B_{0j} = bB_{0,j-1} \qquad \text{for } 0 < j \leq c$$

$$A_{i0} = aA_{i-1,0}$$

$$B_{i0} = aB_{i-1,0} + bA_{ic} + bB_{ic} \qquad \text{for } 0 < i \leq c$$

$$A_{00} = aA_{c0} + aB_{c0}$$

$$B_{00} = bA_{0c} + bB_{0c}$$

By adding pairs of equations, we can verify $A_{ij} + B_{ij} = \frac{1}{(c+1)^2}$. This corresponds to the fairly obvious fact that asymptotically, every pair of counts (without regard to the order of the list) is equally likely.

Substituting this relation gives

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1} \qquad \text{for } 0 < i,j \leq c$$

$$A_{0j} = bA_{0,j-1} + \frac{a}{(c+1)^2} \qquad \text{for } 0 < j \leq c$$

$$A_{i0} = aA_{i-1,0} \qquad \text{for } 0 < i \leq c$$

$$A_{00} = \frac{a}{(c+1)^2}$$

This is equivalent to the system

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1} \qquad \text{for } 0 \leq i,j \leq c$$

$$A_{-1,j} = \frac{1}{(c+1)^2} \qquad \text{for } 0 \leq j \leq c$$

$$A_{i,-1} = 0 \qquad \text{for } 0 \leq i \leq c$$

For convenience, extend these recurrences to hold for all $i,j \geq 0$. This will not effect the A_{ij} we are interested in. The recurrence can now be solved by the use of generating functions. Define

$$F(x,y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} A_{ij}x^iy^j$$

$$= \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} (aA_{i-1,j} + bA_{i,j-1})x^iy^j$$

$$= a\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} A_{i-1,j}x^iy^j + b\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} A_{i,j-1}x^iy^j$$

$$= ax\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} A_{ij}x^iy^j + a\sum_{j=0}^{\infty} A_{-1,j}y^j + b\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} A_{ij}x^iy^j$$

$$= ax\,F(x,y) + \frac{a}{(c+1)^2(1-y)} + by\,F(x,y)$$

Solving for F(x,y) gives

$$F(x,y) = \left(\frac{a}{(c+1)^2(1-y)}\right)\left(\frac{1}{1-ax-by}\right)$$

$$= \frac{a}{(c+1)^2}\left(\sum_{i=0}^{\infty} y^i\right)\left(\sum_{j=0}^{\infty} (ax+by)^j\right)$$

Using the binomial theorem gives

$$= \frac{a}{(c+1)^2}\left(\sum_{i=0}^{\infty} y^i\right)\left(\sum_{j=0}^{\infty}\sum_{k=0}^{j} \binom{j}{k}(ax)^{j-k}(by)^k\right)$$

$$= \frac{a}{(c+1)^2}\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\sum_{k=0}^{j} \binom{j}{k}a^{j-k}b^kx^{j-k}y^{i+k}$$

Now substitute i' for j-k and j' for i+k and then drop the primes.

$$= \frac{a}{(c+1)^2}\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\sum_{k=0}^{j} \binom{i+k}{k}a^ib^kx^iy^j$$

Therefore $$A_{ij} = \frac{a}{(c+1)^2}\sum_{k=0}^{j} \binom{i+k}{k}a^ib^k$$

Prob(A ahead of B) is then $\sum_{i=0}^{c} \sum_{j=0}^{c} A_{ij}$

$$= \frac{a}{(c+1)^2} \sum_{i=0}^{c} \sum_{j=0}^{c} \sum_{k=0}^{j} \binom{i+k}{k} a^i b^k$$

$$= \frac{a}{(c+1)^2} \sum_{k=0}^{c} \sum_{i=0}^{c} \binom{i+k}{k} a^i b^k \sum_{j=k}^{c} 1$$

$$= \frac{a}{(c+1)^2} \sum_{k=0}^{c} (c-k+1) b^k \sum_{i=0}^{c} \binom{i+k}{k} a^i$$

Recalling that a and b were originally $\frac{a}{a+b}$ and $\frac{b}{a+b}$ and substituting into the cost formula finishes the proof. □

Another interesting fact about this rule is that for some distributions we can prove that it does not approach the optimum as we increase the number of bits.

Theorem: Given a distribution of key request probabilities, if $p_j < p_i < 2p_j$ for some i and j, the wait c and move to front rule will not approach the optimum as $c \to \infty$.

Proof: We show that Prob(k_i ahead of k_j) does not approach 1 as $c \to \infty$, hence the cost is bounded away from the optimum.

For convenience, let $a = \frac{p_i}{p_i+p_j}$ and $b = \frac{p_j}{p_i+p_j}$. From the preceeding theorem, Prob (k_i ahead of k_j) =

$$\frac{a}{(c+1)^2} \sum_{k=0}^{c} (c-k+1) b^k \sum_{i=0}^{c} \binom{i+k}{k} a^i$$

$$< \frac{a}{(c+1)^2} \sum_{k=0}^{c} (c-k+1) b^k \sum_{i=0}^{\infty} \binom{i+k}{k} a^i$$

$$= \frac{a}{(c+1)^2} \sum_{k=0}^{c} (c-k+1) b^k \frac{1}{(1-a)^{k+1}}$$

Since $1-a = b$,

$$= \frac{a}{b(c+1)^2} \sum_{k=0}^{c} (c-k+1)$$

$$= \frac{a}{b(c+1)^2} \left[(c+1)^2 - \frac{c(c+1)}{2}\right]$$

$$= \frac{a}{b} \cdot \frac{c+2}{2c+2}$$

which approaches $\frac{a}{2b} = \frac{p_i}{2p_j}$ as $c \to \infty$. Since $p_i < 2p_j$, $\lim_{c\to\infty}$ Prob(k_i ahead k_j) $= \frac{p_i}{2p_j} < 1$ and the cost cannot approach the optimum as $c \to \infty$. □

Indeed, it is reasonable to expect the theorem to hold for all distributions except the uniform and the distribution with a key of probability one. However, this conjecture has not yet been proved.

It is interesting to determine why this method decreases the cost over the move to front rule. The wait c, move and clear rule achieved a decrease by altering the probability that a key is moved from the request probabilities to a more favorable distribution. However, the wait c and move rule does not do this. Since a key is moved after every $(c+1)^{st}$ request for it, the move probabilities remain unchanged in the sense that a key requested with probability p_i will account for a fraction, equal to p_i, of the total number of moves.

Consider any two keys, k_i and k_j. If we assume that moves occur at intervals which are independent of whether or not k_i is ahead of k_j, then k_i will be ahead $\frac{p_i}{p_i+p_j}$ of the time and the performance will be

the same as the move to front rule. However, this is not the case. After k_i has been moved (assume $p_i > p_j$), its count is set to zero. Asymptotically, k_j's count is uniformly distributed over $\{0,1,\ldots,c\}$. After k_j has been moved, its count is zero, and k_i's count ranges from zero to c. Clearly, after k_j has been moved, the next move will occur sooner; the roles of k_i and k_j have been interchanged, and after k_j has been moved, the count of k_i (the more probable key) is closer to causing a move. Therefore, the probability we find k_i ahead of k_j is increased because we must wait longer for the next move when it is ahead of k_j.

Finally, we notice that this rule will have much faster convergence than the wait c, move and clear rule since on the average, it will move a key after every c+1 requests. The performance of this rule is compared with previous rules in Table 2.9.1.

Having analyzed these rules, we can see that they are asymptotically inferior to both the wait c, move and clear rules and the limited difference rule. The convergence is faster than the wait c, move and clear rules. It is at most c+1 times slower than the corresponding permutation rule, while the wait c, move and clear rule may be as bad as nc+1. A final interesting fact is that for some probability distributions, it can be proved that this rule <u>does not</u> approach the optimum as $c \to \infty$. We conjecture this to hold for any probability distribution except the uniform and the distribution with a key of probability one.

Table 2.9.1

Comparison of Rules that use Counters

	c=0	c=1	c=2	c=3	c=4	c=5
Limited Difference Rule	3.9739	3.4162	3.3026	3.2545	3.2288	3.2113
Wait c, Move to Front and Clear (Exact)	3.9739	3.6230	3.4668	3.3811	3.3285	
Wait c, Transpose and Clear (Exact)	3.4646	3.3399	3.2929	3.2670	3.2501	
Wait c and Move to Front (Exact)	3.9739	3.8996	3.8591	3.8338	3.8165	3.8040
Wait c and Transpose	3.4646	3.3824	3.3576	3.3473	3.3312	3.3272

Asymptotic costs for various rules assuming a nine element list whose probabilities are given by Zipf's Law. Compare these with the optimal cost which is 3.1814. Cost for the limited difference rule and the wait c and transpose rule were estimated by simulations consisting of 1000 requests. The average of 200 trials is shown.

2.10 Time Varying Distributions

In this section, we consider probability distributions that vary with respect to time. We first examine two examples concerning the move to front rule: one where the probability of a key decreases after it has been requested, and another where it increases.

The first example supposes we have n keys, $k_1, k_2, \ldots, k_n$. Assume the requests made to this list form a sequence of permutations of these n keys. The permutations are independently chosen with each of the n! permutations being equally likely. A model that satisfies this constraint is a company that sends out bills each month. Its customers then pay their bills in a random order.

Assuming this model, we can prove that the move to back rule is the optimal rule. The proof is as follows: after t requests out of a permutation have been made, each of the remaining n-t requests is equally likely, and the best we can do is to have these n-t keys (and none of the t previously requested keys) in the first n-t positions of the list. Since each key is equally likely to be requested, the ordering of the unrequested keys will make no difference. The move to back rule clearly achieves this and therefore must be optimal. Any other rule will occasionally move the requested key to one of the first n-t positions, resulting in a higher cost. To derive the average cost for the move to back rule to retrieve all n keys of a permutation, we note that to retrieve the i^{th} key, we search through an unordered list of n-i+1 keys. The average cost is then

$$E(Cost_{MTB}) = \frac{1}{n} \sum_{i=1}^{n} \frac{(n-i+1) + 1}{2} = \frac{n+3}{4}$$

If no rule is applied to the list, each key will be accessed exactly once giving a cost of

$$E(Cost_{RAND}) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}.$$

Finally if the move to front rule is used, accessing the list at time i will first require i-1 comparisons with the previously requested keys. Then we search through an unordered list of n-i+1 keys. The cost is then

$$E(Cost_{MTF}) = \frac{1}{n} \sum_{i=1}^{n} (i-1) + \frac{(n-i+1) + 1}{2} = \frac{3n+1}{4} .$$

This cost is three times larger than the move to back rule, and 50 percent larger than doing no moves at all. The reason is obvious: once a key has been requested, its probability of being requested again decreases. In this case, our strategy must be to move requested keys back in the list.

Using the move to back rule, the keys will appear in the list in the order that they were requested. If our clients have regular habits and pay their bills at about the same time each month, the access time of the move to back rule will decrease further, and that of the move to front rule will increase.

We now consider a second example. Suppose that with probability p, the requested key is the same as the previously requested key.

With probability 1-p some distribution $(p_1,p_2,\ldots,p_n)$ over the keys is used. The move to front rule would seem to be the logical choice here (if p is not extremely small) since the first key in the list will have a good chance of being requested again.

To analyze this rule, we note that the probability of a given ordering is not effected by p, but depends only on the p_i. We can view the chain as waiting in each state until a "normal" request is made. During this wait, only requests to the first key are made, and these do not change the order of the list. In addition, the wait time is the same for all states.

Therefore, with probability p, the first key is found in one comparison. With probability 1-p, the $p_1,p_2,\ldots,p_n$ distribution is used. The cost here is just $1 + 2 \sum_{1\leq i<j\leq n} \frac{p_i p_j}{p_i+p_j}$, the normal move to front cost. Adding these two results gives

$$p \cdot 1 + (1-p) \cdot [1 + 2 \sum_{1\leq i<j\leq n} \frac{p_i p_j}{p_i+p_j}]$$

$$= 1 + 2(1-p) \sum_{1\leq i<j\leq n} \frac{p_i p_j}{p_i+p_j} \quad ,$$

a decrease of nearly a factor of 1-p. If p is close to one, the move to front rule gives very good performance.

These two examples point out much of the performance depends on the model for the input requests. For models that cause the requested key to become more probable, the move to front rule will perform well. If the requested key becomes less probable, a rule that moves the requested

key back in the list is more suitable. Due to the wide variety of performance, little more can be said unless a specific model is considered.

2.11 Summary and Conclusion

We first discussed the move to front rule and the transposition rule. Asymptotically, the transposition rule performs better. Rivest [2] has shown that for any distribution its asymptotic cost is less than or equal to that of the move to front rule. We calculated the asymptotic cost for several distributions, with the transposition rule showing about a 10 percent increase over the optimum, and the move to front rule a 25-38 percent increase. Finally, a theorem by Rivest [2] showed that these costs could be at most twice the optimum. Thus, if we expect the number of requests to be large compared to the number of keys, the asymptotic cost will dominate and the transposition rule will be superior.

Asymptotic cost is not the only criterion for evaluating rules. A rule may have very low asymptotic cost, but converge so slowly that it is of little practical value. We defined the overwork in order to measure the speed of convergence. The move to front rule was found to have much smaller overwork. For two simple distributions, the move to front overwork was $\frac{n-1}{2}$ (for an n element list), compared to $\frac{n^2-1}{6}$ for the transposition rule. For Zipf's Law, the move to front rule has overwork $\approx .057n^2$, while the transposition rule has $\Omega(n^3)$ overwork.

The difference in rates of convergence was also demonstrated by graphs of the time varying cost. From these, we calculated when the

total cost of the transposition rule would become less than that of the move to front rule. For Zipf's Law, it appears to take $\Omega(n^2)$ requests before the crossover occurs. Thus, if the number of requests will be small ($O(n^2)$), compared to the number of keys, the move to the front rule outperforms the transposition rule.

We next considered a subset of permutation rules called move ahead k rules. These rules form a spectrum ranging from the move ahead 1 rule (transposition) to the move ahead n-1 rule (move to front). As the parameter k is increased, the asymptotic cost increases, but the rate of convergence also increases.

A hybrid rule that initially moves keys to the front and then begins transposing was also examined. If the anticipated number of requests is neither large enough to make the transposition rule a clear choice nor small enough to require the move to front rule, the hybrid rule should be used. It was shown to combine the best features of both rules: initially it converges quickly, and asymptotically it has a low cost. Note that it is only in this intermediate region that both fast convergence and low asymptotic cost are important. Outside of this region, the hybrid either performs like either the move to front rule or the transposition rule, and it is better to use these rules than incur the overhead of the hybrid. A difficulty was found in deciding when to switch rules. For Zipf's Law, this point appears to be .268n + .980.

The first request rule and the move to front rule was shown to produce any list with the same probability. Thus, these two rules are

essentially the same, but there are two important differences. First, the first request rule moves each key only once and therefore is much cheaper to execute than the move to front rule. Second, the first request rule will be "trapped" in some ordering after all keys have been requested. Thus, its cost, averaged over time, has a much higher variance than that of the move to front rule. Thus, this rule can be used in place of the move to front rule. The advantage here is that the first request rule is cheaper to execute (each key is moved only once). However, it also has the drawback of increasing this variance.

The asymptotic ordering obtained by the first request rule is the same as if the keys were not known a priori and a "new" key (one requested for the first time) is inserted at the end of the list. Since this is also the steady state distribution for the move to front rule, if the initial list was constructed in this manner, the move to front rule will not reduce the cost.

Finally, a hybrid rule between the first request and transposition rules was formulated. This has better performance than the move to front/transposition hybrid, and also, we need not guess when to switch rules.

In comparing the different rules we have studied, we find that the move to front rule is best if a small number of requests will be made. The transposition rule is best for a large number of requests, and the first request rule/transposition hybrid should be used for an intermediate number. However, none of these rules should be used if

we have storage to keep counters. If this is the case, one of the following methods should be used.

We next considered rules that used counters. The first of these was the frequency count rule. The performance of this rule is optimal at any time and asymptotically. Its major disadvantage is the storage required by the count fields. This can be reduced by storing the differences between successive counts.

The limited difference rule put an upper bound on the size of these differences. The performance is no longer optimal, but approaches the optimum as the upper bound on the differences goes to infinity. This upper bound need not be too large. Even for small bounds, the performance is nearly optimal.

The wait c, move and clear rules improve on the performance of the corresponding permutation rule by altering the probability that a key is moved. As $c \to \infty$, the performance approaches the optimum. However, the performance of the limited difference rule is better and these rules also have the disadvantage of converging very slowly.

A final class of rules was the wait c and move rules. These rules also improved upon their corresponding permutation rule, but the cost does not approach the optimum as $c \to \infty$, and these rules were outperformed by both the limited difference rule and the wait c, move and clear rules.

In comparing the different rules using counters, the frequency count rule should be the choice if enough storage can be spared for the

counters so that there is no possibility of overflow. If this is not the case, the limited difference rule appears to be the best. Its asymptotic cost is the lowest, and it does not have the slow convergence of either the "wait" rules.

Finally, we considered the effects of a time varying distribution. In one example, the move to back rule is optimal, and the move to front rule is poorer than simply leaving the list unchanged. In another example, the move to front rule performed quite well. The difference is that once a key has been requested in the first example, its probability of being requested decreases. In the second example, it increases.

3. BINARY SEARCH TREES

In this chapter, we will discuss extensions of the previously discussed techniques to binary search trees. The standard definitions given in Knuth [3] will be used. Note that the cost function is still $\sum_{i=1}^{n} p_i c_i$, but now c_i is the level of k_i (with the root having level one.) Here it is necessary to assume that there is an ordering imposed on the keys. The tree search algorithm requires that every node in the tree must be greater than every node in its left subtree and less than every node in its right subtree. Any transformation we perform on the tree must preserve this property.

Results that are related to this topic fall into two categories. The first assumes that key request probabilities are known a priori. If this is the case, an algorithm by Knuth [11] can be used to determine the optimal binary tree. Heuristic algorithms that build near-optimal trees, but require less space and time have been discovered by Bruno and Coffman [12], Melhorne [13], and Walker and Gottlieb [14].

The second category of results contains the height balanced trees of Adelson-Velskii and Landis [15], and the weight balanced trees of Nievergelt and Reingold [16]. These methods balance the tree when keys are inserted and deleted. No changes are made if a key already in the tree is requested, so these methods do not take advantage of a favorable probability distribution by moving more frequently accessed keys nearer the root. The methods we will examine do not fall into either category since we assume the probabilities are <u>not</u> known a priori,

and we will suppose that there are high probability keys which we wish to move near the root of the tree.

We will study the two transformations shown below, called "rotations."

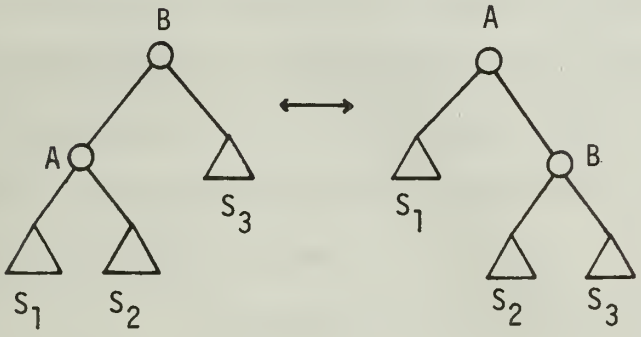


Figure 3. The two rotations.

Here the circles labeled A and B are nodes, and the triangles labeled S_1, S_2 and S_3 are subtrees. Note that this transformation can be performed at any node in the tree. This pair of transformations is acceptable since it does preserve the ordering of the tree; after a rotation, the left subtree of A contains only keys less than A, and the right subtree contains only those greater, and similarly for B. This pair of transformations is also "complete" in the following sense:

Theorem: Let T_1 and T_2 be two binary search trees that have the same set of keys. Then T_1 can be transformed into T_2 by a sequence of rotations.

Proof: Let T_1 be the root of T_2. We can bring r to the root of T_1 by using rotations to successively promote it until it reaches the root.

Since rotations preserve the ordered property of the tree, the nodes in r's left subtree will be less than r, and the nodes in its right subtree will be greater. We then recursively apply this procedure to each subtree to generate first the left and then the right subtrees of T_2. Note that the transformations applied to the right subtree will leave the left subtree unchanged. Hence, this procedure successfully produces T_2. □

We consider these transformations as a mechanism to move node A and subtree S_1 to higher positions in the tree when we suspect they contain high-probability nodes. The important question is: when should the transformation be used? The following sections describe several different rules for using the transformations.

3.1 Transform after Every Request

We first consider rules analogous to the transposition and move to front heuristics. The move to root rule uses rotations to repeatedly promote the requested node until it becomes the root of the tree. The move up one rule uses a rotation to promote the requested node one level.

Although it is not immediately obvious, the operation of moving a node to the root can easily be done during the search for the requested key. Suppose x has been requested and has subtrees S_L and S_R. Let $\ell_1,\ldots,\ell_i$ be the ancestors of x which are less than x (labeled in order of distance from the root) and suppose they have left subtrees $S_{\ell_1},\ldots,S_{\ell_i}$. Similarly, let $r_1,\ldots,r_j$ be the ancestors greater than x with right subtrees $S_{r_1},\ldots,S_{r_j}$. We can then construct a tree with x as its root as shown below.

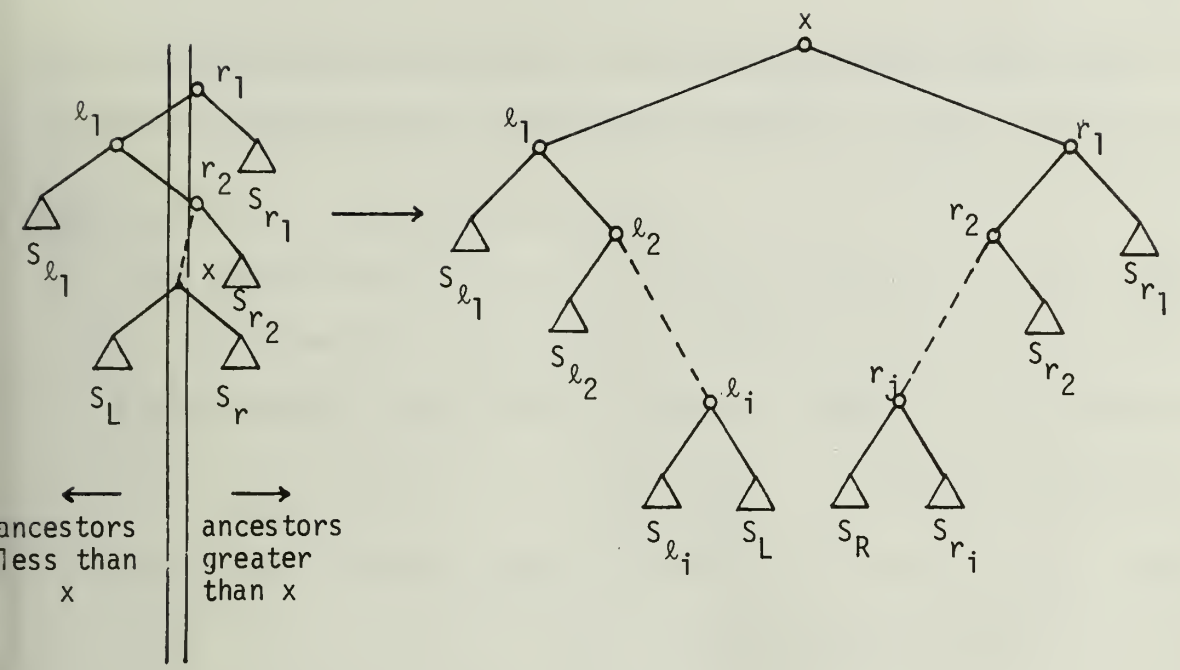


Figure 3.1.1 Moving node x to the root.

To accomplish this transformation during the search, we simply keep a list of the ancestors of x which are less than x using their right pointers and a list of those greater than x using their left pointers. When x is found, his sons become ℓ_1 and r_1, the heads of these two lists. S_L becomes the right subtree of ℓ_i, and S_R becomes the left subtree of r_j. The case where x has no left (or right) ancestors is easily handled by a few tests.

To analyze the performance of the move to root rule, we introduce the first request rule. The first time a key is requested, it is promoted in the tree until it reaches the root or becomes the son of previously requested key. The resulting tree is the same as the one obtained by inserting a "new" key (one requested for the first time) into

the tree. As in the case of the linked list, the move to root rule and the first request rule behave identically. To prove this, we first make three observations.

Observation 1: Consider a tree that is modified by the move to root rule. For any two keys, x and y, if a key which is between x and y in the ordering on the keys is requested, x will not be an ancestor of y in the resulting tree.

Proof: Since the root of the resulting tree is between x and y, they will be in different subtrees of the root. □

Observation 2: If x is an ancestor of y and a key that is not between x and y is requested, x will still be an ancestor of y.

Proof: Suppose that z has been requested. If z is not a descendant of x, the tree rooted at x will not be altered, so x will still be an ancestor of y. If z is a descendant of x, it will be moved up in the tree until it becomes a son of x. There are two cases

Case (1): $z < x$. The rotation looks like:

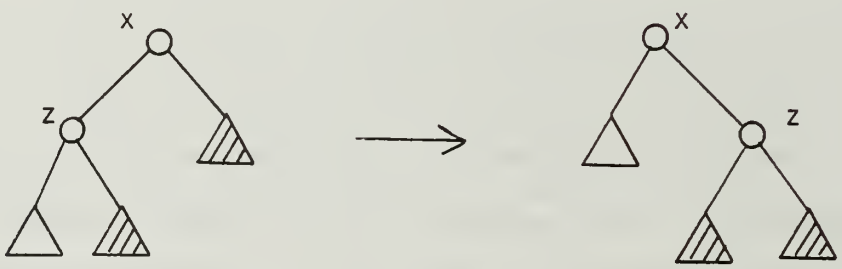


Figure 3.1.2

where y is in either of the shaded subtrees. Note that y cannot be in the left most subtree since then z would be between x and y.

Case (2): z > x. The transformation is:

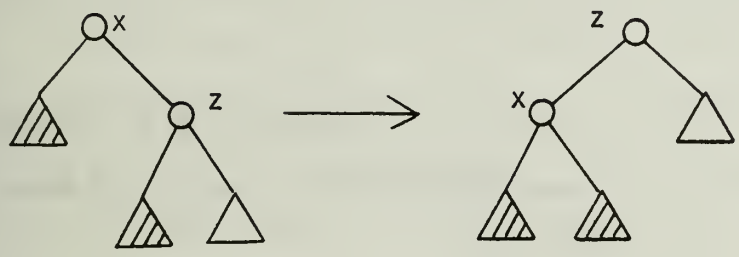


Figure 3.1.3

where, again y must be in either shaded subtree.

In either case, x is still an ancestor of y. Since z is no longer a descendant of x, any further rotations will leave x as an ancestor of y, and therefore Observation 2 must be true. □

Observation 3: If neither x nor y is the ancestor of the other and a key that is not between x and y is requested, then neither x nor y will become the ancestor of the other.

Proof: If neither x nor y is the ancestor of the other, there exists some w that is between x and y, and an ancestor of both. Since z is not between x and y, it cannot be between x and w and it cannot be between y and w. By Observation 2, w will still be an ancestor of both x and y in the resulting tree and hence neither x nor y will become the ancestor of the other. □

We now give a lemma that characterizes exactly when one node will be an ancestor of another, based on the sequence of requests and the initial tree.

Lemma 1: Node x will be an ancestor of node y using the move to root rule if and only if:

(1) Neither x, nor y, nor any key between them in ordering on the keys has been requested, and x was an ancestor of y in the initial tree.

OR (2) Neither y nor any key between x and y has been requested after the most recent request for x.

Proof: ("if" part)

(1) => Lemma follows from Observation 2.

(2) => Lemma follows from Observation 2 and the fact that when x is requested, it becomes the root of the tree and hence is an ancestor of every other node.

("only if" part)

Case 1 (x has not been requested)

Suppose that x has not been requested, and it is an ancestor of y. We will show that this must imply (1). From the observations it is clear that the only way x can become an ancestor of y (if it is not already) is for x to be requested. Since x was never requested, it must have originally been an ancestor of y. Then, from Observation 2, no key between x

and y can have been requested since then x would no longer be an ancestor of y. Similarly, y cannot have been requested. Therefore (1) holds.

Case 2 (x has been requested)

Here we show (2) must hold. Consider the situation after the most recent request for x: x is an ancestor of y, and x will not be requested again. This is the same situation as in Case 1 and by using its proof, we can show (2) must hold. □

We also prove the following lemma about the first request rule.

Lemma 2: Node x will be an ancestor of node y using the first request rule if and only if:

(1) Neither x nor y nor any node between them has been requested and x was an ancestor of y in the original tree.

OR (2) Neither y nor any node between x and y was requested before the first request for x.

Proof: Case 1 (x has not been requested.)

First note that the three observations still hold if x has not been requested. Then, as we noted before, once the requested node (z) is no longer a descendant of x, further rotations involving z do not effect the tree rooted at x. Hence the

two rules "look the same" to an unrequested x because the only differences occur after z is no longer a descendant of x. Therefore the proof for Lemma 1 is valid and Case 1 is proved.

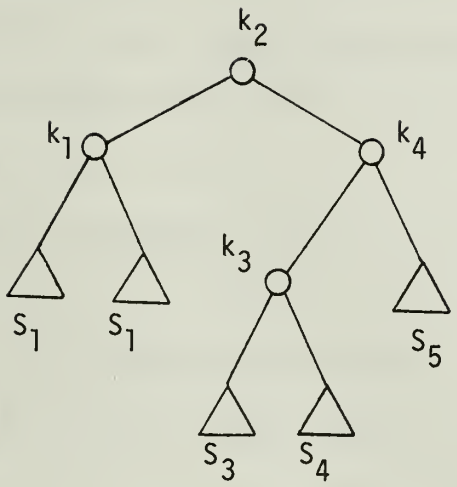
Case 2 (x has been requested)

To see what happens when x is first requested, consider the previously requested keys and label them $k_1, k_2, \ldots, k_n$ so that $k_1 < k_2 < \ldots < k_n$. They occur in a group at the top of the tree and divide the unrequested nodes into n+1 different subtrees. The leftmost of these subtrees contains all keys less than k_1, and the rightmost contains all greater than k_n. Each of the remaining consist of all keys between two "adjacent" k_i. (See Figure 3.1.4)

Thus, two unrequested nodes, x and y, are in the same subtree if and only if no key between them has been requested. When x is first requested, it moves to the root of the subtree it is in and becomes an ancestor of all nodes in that subtree. Therefore x becomes the ancestor of y if and only if neither y nor any key between x and y has been requested. x will then remain the ancestor of y since no node can move up past x and out of its subtree, proving Case 2. □

We can now prove the main theorem.

Theorem: Given any initial tree, the probability of obtaining a given final tree after any number of requests is the same for the move to root rule and the first request rule.



Keys k_1,k_2,k_3 and k_4 have been requested. S_1 contains all keys less than k_1. S_5 contains all keys greater than k_4. S_i contains all keys between k_{i-1} and k_i for $i = 2,3,4$.

Figure 3.1.4 How the requested keys divide the tree.

Proof: Consider any sequence of requests $r_1, r_2, \ldots r_k$ as inputs to the move to root rule and the reversed sequence $r_k, r_{k-1}, \ldots, r_1$ as inputs to the first request rule. Note that these two sequences have the same probability. Trivially, the conditions of Lemma 1 hold if and only if the conditions to Lemma 2 hold. This means that x is an ancestor of y in one tree if and only if it is an ancestor of y in the other. Since this information allows us to uniquely construct a tree, the two trees are the same and the theorem is proved. □

Note also that the theorem also holds if we are given a probability distribition over the initial trees since the two rules perform identically on each tree.

As is the case with linked lists, the first request rule creates the same tree as if the keys were not known a priori, and each "new" key (one requested for the first time) was inserted into the tree. If the initial tree was created in this manner, the move to root rule will not decrease the cost.

The characterization given in Lemma 1 allows us to determine the time varying and steady state costs for the move to root rule. As stated in the theorem, these will equal the cost of the first request rule.

Theorem: If key k_i has probability p_i of being requested and the keys are ordered $k_1 < k_2 < \ldots < k_n$, the cost for the move to root rule after t requests is:

$$1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{B_{ij}} + \sum_{1 \le i < j \le n} [p_i A_{ji} + p_j A_{ij} - \frac{2p_i p_j}{B_{ij}}](1-B_{ij})^t$$

Where A_{ij} is the probability that k_i is an ancestor of k_j in the initial tree and $B_{ij} = \sum_{k=\min(i,j)}^{\max(i,j)} p_k$, the probability of requesting a key between k_i and k_j inclusive.

Proof: We first determine Prob(k_i is an ancestor of k_j after t requests). By Lemma 1, this is the sum of:

(1) The probability that neither k_i nor k_j nor any key between them has been requested in t requests and k_i was originally an ancestor of k_j. This probability is $(1-B_{ij})^t A_{ij}$.

and (2) The probability that neither k_j nor a key between k_i and k_j has been requested since k_i's most recent request. This equals the probability that k_i was most recently requested at time m and neither k_j nor any key between k_i and k_j nor k_i (since m was the most recent request) was requested after time m. Hence the probability of (2) equals

$$\sum_{m=1}^{t} p_i (1-B_{ij})^{t-m} = p_i \frac{1-(1-B_{ij})^t}{B_{ij}}$$

Therefore Prob(k_i is an ancestor of k_j at time t) =

$$(1-B_{ij})^t A_{ij} + p_i \frac{1-(1-B_{ij})^t}{B_{ij}}$$

Then

$$E(\text{Cost}) = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \text{Prob}(k_j \text{ is an ancestor of } k_i \text{ at time } t)$$

$$= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{B_{ij}} + \sum_{1 \leq i < j \leq n} [p_i A_{ji} + p_j A_{ij} - \frac{2p_i p_j}{B_{ij}}](1-B_{ij})^t$$

□

It is interesting to note that the asymptotic cost equals $1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{B_{ij}}$, which bears a striking resemblance to the asymptotic cost of the move to front rule (p_i+p_j has been replaced by B_{ij} in the denominator). Also, the formula gives the initial cost (t=0) as

$$1 + \sum_{1 \leq i < j \leq n} [p_i A_{ji} + p_j A_{ij}].$$

For a tree built by random insertion, k_i will be the ancestor of k_j ($i<j$) if and only if it is the first to be inserted from the set of all keys from k_i to k_j inclusive. Since each of these $j-i+1$ keys is equally likely to be first, this gives us $A_{ij} = A_{ji} = \frac{1}{j-i+1}$, and the cost equals

$$(\sum_{i=1}^{n} p_i [H_i + H_{n-i+1}]) - 1.$$

We now consider the move up one rule. We would expect it to have lower asymptotic cost than the move to root rule (as an analogy to the case of linked lists). However, simulations (See Table 3.1.1) suggest that this is not the case. The move to root rule had significantly lower asymptotic cost in four out of six distributions tested. In addition its average asymptotic cost was lower.

Table 3.1.1

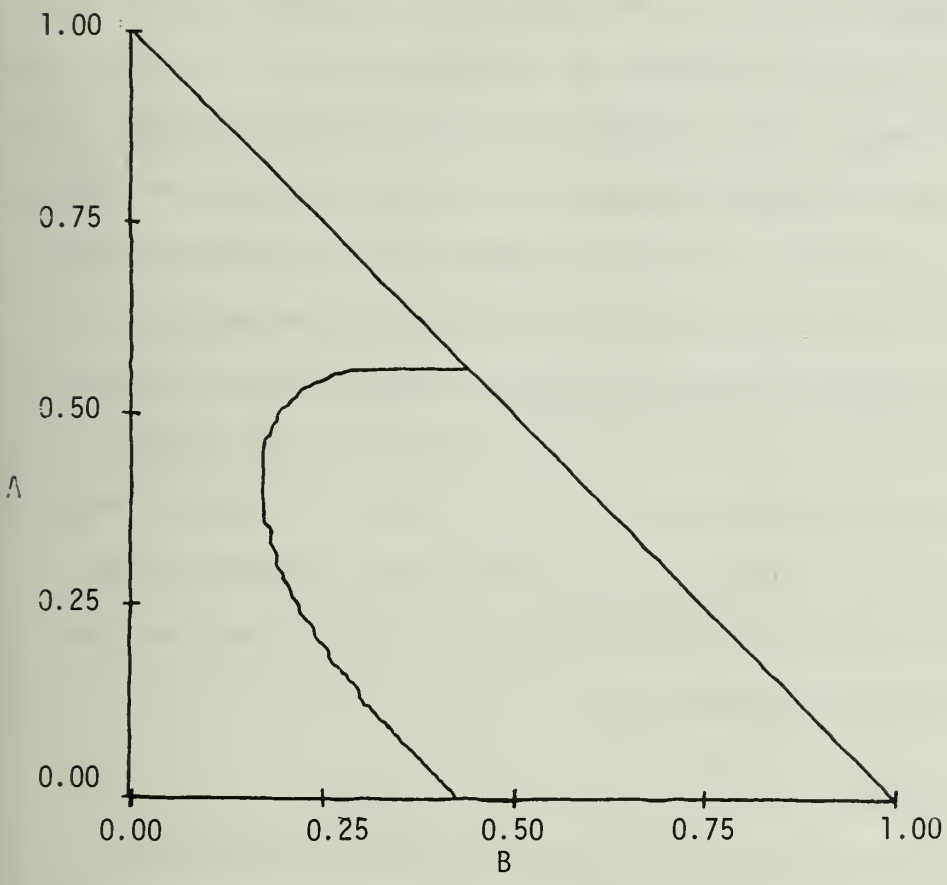
A Comparison of the Move to Root and Move Up One Rules

	English Letters	ZIPF'S LAW #1	#2	#3	#4	#5	Average
Random Cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal Cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
MTR Cost (Exact)	4.31	5.63	5.53	5.68	5.59	5.55	5.59
Increase Over Optimum	29.8%	37.0%	40.7%	36.5%	37.6%	40.0%	38.4%
Move Up One Rule Cost	4.77	6.27	5.52	6.07	6.18	5.43	5.90
Increase Over Optimum	43.8%	52.8%	40.6%	45.9%	52.1%	37.1%	45.7%

A simulation was run to determine the cost of the move up one rule (and other rules appearing in later tables). Fifty trees were randomly generated, and the cost and other statistics were recorded after 500 requests. The probability distributions we considered were the English letters and five others that were generated by choosing a random ordering of 100 keys whose probabilities were given by Zipf's Law.

Further evidence is provided by considering these rules applied to a tree of only three nodes. We label the keys A, B and C with probabilities a, b and c respectively. The rules form Markov chains with five states, each corresponding to a different tree of three nodes. The transition matrices are easily determined, and from them, the steady state cost is easily obtained. This calculation was done for a = 0, .01, .02,...,.99, 1. b = 0, .01,...1-a and c = 1-a-b. The results are shown in Figure 3.1.5. Note that the move to root rule does outperform the move up one rule for a considerable number of the data points. Since these calculations were not simulations, but were done exactly (within the precision of the computer), the move to root rule does have a lower asymptotic for some distributions, and hence a theorem showing the move up one rule to always be superior (as for linked lists) cannot be true.

To get an intuitive idea of why the move up one rule can behave poorly, let us consider what will cause a given node to move up in the tree. For a given node B, let us look at A_1, A_2,...,A_k, the ancestors of B, ordered in increasing distance from the root. From the properties of the rotations, we can verify the B will move up in the tree if B itself is requested or if A_i is requested and $A_{i-1} < A_i < B$ or $A_{i-1} > A_i > B$. B will move down one level if either of its sons is requested or the son of A_i that is not A_{i+1} is requested. Thus, we can see that the movements of B are controlled by much more than just its probability. If B is far from the root, it may be difficult for B to



The curve in this figure shows those a and b where the cost of the move to root rule equals that of the move up one rule. The move to root rule has lower cost in the region to the right of the curve (53% of the total area) and the move up one rule has lower cost in the region to the left.

Figure 3.5.1

move up in the tree. On the other hand, the move to root rule promotes nodes all the way to the root of the tree, so high probability nodes cannot spend a lot of time "trapped" far from the root.

We derived a closed form for the cost of the move to root rule as a function of time which bore a striking resemblance to that of the move to front rule. The move to root rule was also shown to be identical to the first request rule. A simulation estimating the cost of the move up one rule suggested it was often inferior to the move to root rule (see Table 3.1.1). Both rules performed well and provided reasonable decreases over the cost of a random tree. The move to root rule averaged within 38 percent of the optimum, while the move up one rule was within 45 percent. These average costs suggest that the move to root rule would be the better choice.

3.2 Monotonic Trees

Another method for getting more frequently accessed nodes high in the tree is to keep a frequency count associated with each node. The node with the largest count becomes the root of the tree, and each subtree is formed recursively, using the same rule. Such a tree is called monotonic because the frequency count for any given node is greater than or equal to that of any of its descendants. (This property is the same as the one required for a heap, see Williams [17].)

It is a simple matter to keep the tree ordered in this manner. Rotations are used to promote the requested key until a key with equal or greater count is encountered. The resulting tree will have the monotonic property.

Asymptotically, the most probable key will become the root of the tree (by the Law of Large Numbers, it will be requested the most times), and each subtree will have its most probable node as its root. The asymptotic tree will be monotonic, with probabilities as weights. This allows us to easily calculate the asymptotic cost for this method. Table 3.2.1 show it averages within 15 percent of the optimum.

However, this method is very poor for some distributions. Suppose key k_i has probability p_i and that the lexicographic ordering of the keys is $k_1 < k_2 < \ldots < k_n$.

Table 3.2.1

The Performance of Monotonic Trees

		ZIPF'S LAW					
	English Letters	#1	#2	#3	#4	#5	Average
Random Cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal Cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
Monotonic Tree Cost (Exact)	3.77	4.91	4.18	5.32	4.68	4.20	4.66
Increase Over Optimal	13.6%	19.7%	6.5%	27.9%	15.1%	6.0%	15.1%

See Table 3.1.1 for explanation.

If the p_i are approximately equal and $p_1 > p_2 > \ldots > p_n$, then the skewed tree shown below will result.

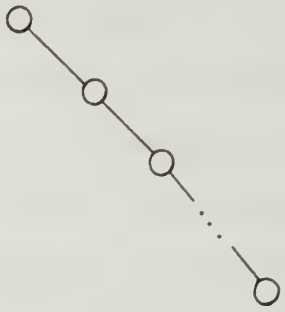


Figure 3.2.1 A worst case monotonic tree.

A theorem by Melhorne [13] shows how bad this can be.

Theorem (Melhorne [13]): The ratio between the cost of a monotonic tree and the optimal tree may be as high as $n/(4 \log n)$ for trees with n nodes.

This theorem depends on a very unfavorable choice for the ordering of the keys and only gives an idea of the worst case performance of monotonic trees. We now consider how these trees perform on the average by assuming the probabilities are randomly chosen in some way. The first method we consider chooses the probabilities from a given set of n probabilities. The second chooses the probabilities from some given probability density function. We now investigate the first method.

Theorem (Knuth [3, p. 432]): Given n keys and n probabilities ($p_1 \geq p_2 \geq \ldots \geq p_n$), if each of the n! assignments of probabilities to keys is equally likely, the expected cost of a monotonic tree is $[2 \sum_{i=1}^{n} H_i p_i] - 1$.

Proof: An assignment of probabilities to the keys imposes an ordering on the probabilities. Probability p_i is to the left of p_j if the key to which p_i has been assigned is to the left of the key to which p_j has been assigned. We have assumed that each of the n! orderings is equally likely. The cost of a monotonic tree is solely determined by this ordering imposed on the probabilities. Hence, the problem is equivalent to assigning p_i to key k_i and then randomly ordering the k_i since each of the n! orderings on the probabilities will still be equally likely. This restatement turns out to be simpler, and we work with it instead.

Let ℓ_i be a random variable denoting the level of k_i. By definition,

$$\text{Cost} = \sum_{i=1}^{n} p_i \ell_i$$

So $$E(\text{Cost}) = E\left(\sum_{i=1}^{n} p_i \ell_i\right) = \sum_{i=1}^{n} p_i E(\ell_i).$$

Define $$R_j = \begin{cases} 1 \text{ if } k_j \text{ is an ancestor of } k_i \\ 0 \text{ otherwise} \end{cases}$$

Then $$\ell_i = R_1 + R_2 + \ldots + R_{i-1} + 1 \quad (R_j = 0 \text{ if } j \geq i)$$

$$E(\ell_i) = E(R_1) + E(R_2) + \ldots + E(R_{i-1}) + 1$$

$$= \sum_{j=1}^{i-1} \text{Prob}(k_j \text{ is an ancestor of } k_i) + 1$$

To determine this probability, we discuss some properties of a random ordering. Consider any two keys, k_i and k_j. There are only two

distinct orderings of k_i and k_j; k_i to the left of k_j and k_i to the right of k_j, each having probability 1/2. For either of these two orderings, a third key can be in three different regions: to the left of both keys, between the two keys, and to the right of both keys, each with probability 1/3. In general, any ordering of i keys creates i+1 regions, each having probability 1/i+1 of containing a given key.

Consider any ordering of $k_1,\ldots,k_{j-1}$ and k_i. Now k_j will be an ancestor of k_i if no key with probability greater than k_j (that is, $k_1,k_2,\ldots,k_{j-1}$) occurs between k_j and k_i. For this to happen, k_j must occur in either the region to the left of k_i or the region to the right. Since there are j keys in the ordering, this probability is $\frac{2}{j+1}$.

$$\text{Hence } E(\ell_i) = \left(\sum_{j=1}^{i-1} \frac{2}{j+1}\right) + 1$$

$$= 2H_i - 1$$

$$\text{and hence } E(\text{Cost}) = \sum_{i=1}^{n} p_i(2H_i - 1)$$

$$= \left(2\sum_{i=1}^{n} H_i p_i\right) - 1 \qquad \square$$

The following theorem tells us the cost of a tree built by a random sequence of insertions.

Theorem: Given n keys ($k_1 < k_2 < \ldots < k_n$) and a set of n probabilities $\{p_i : 1 \leq i \leq n\}$, if the probabilities are randomly assigned to the keys and then a tree is built by a random sequence of insertions, its expected cost will be $2\frac{(n+1)}{n} H_n - 3$ for any set of probabilities.

Proof: Let $p(k_i)$ be random variables denoting the probability chosen for k_i and let ℓ_i denote the level of k_i. As before,

$$\text{Cost} = \sum_{i=1}^{n} p(k_i)\ell_i$$

$$E(\text{Cost}) = E\left(\sum_{i=1}^{n} p(k_i)\ell_i\right) = \sum_{i=1}^{n} E(p(k_i)\ell_i)$$

The insertion sequence (and hence ℓ_i) does not depend on $p(k_i)$. These two random variables are independent and

$$E(\text{Cost}) = \sum_{i=1}^{n} E(p(k_i))\, E(\ell_i) = \frac{1}{n} \sum_{i=1}^{n} E(\ell_i)$$

$$E(\ell_i) = 1 + 1 + \sum_{j \neq i} \text{Prob } (k_j \text{ is an ancestor of } k_i)$$

Now k_j will be an ancestor of k_i if and only if it occurs in the insertion sequence before k_i and any key between k_i and k_j. This probability is $\frac{1}{|i-j|+1}$. Therefore

$$E(\ell_i) = 1 + \sum_{j \neq i} \frac{1}{|i-j|+1}$$

$$= 1 + \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$= 1 + [H_i - 1] + [H_{n-i+1} - 1]$$

$$\frac{1}{n} \sum_{i=1}^{n} E(\ell_i) = \frac{1}{n} \left[\sum_{i=1}^{n} H_i + H_{n-i+1}\right] - 1$$

$$= \frac{2}{n} \sum_{i=1}^{n} H_i - 1$$

$$= \frac{2}{n} \sum_{i=1}^{n} \frac{n-i+1}{i} - 1$$

$$= \frac{2}{n} \sum_{i=1}^{n} \frac{n+1}{i} - \frac{2}{n} \sum_{i=1}^{n} 1 - 1$$

$$= \frac{2(n+1)}{n} H_n - 3 \qquad \square$$

This quantity is the same as that derived by Hibbard [18]. However, he assumed that the keys were equally probable, and our result holds for any set of probabilities as long as they are randomly assigned to the keys.

To compare the monotonic and random tree costs, note that if we substitute $p_i = \frac{1}{n}$ into the formula for the monotonic tree cost, we get exactly the expression for the random tree cost.

$$2\left(\sum_{i=1}^{n} H_i \frac{1}{n}\right) - 1 = \frac{2}{n}\left(\sum_{i=1}^{n} H_i\right) - 1 = \frac{2}{n}\left(\sum_{i=1}^{n} \frac{n-i+1}{i}\right) - 1$$

$$= \frac{2}{n}\left[(n+1)\sum_{i=1}^{n} \frac{1}{i} - \sum_{i=1}^{n} 1\right] - 1 = \frac{2(n+1)}{n} H_n - 3. \quad \text{Clearly,}$$

$p_i = \frac{1}{n}$ is the worst case since $p_1 \geq p_2 \geq \ldots \geq p_n$ and the coefficients of the p_i increase with i. Hence, except for the case where $p_i = \frac{1}{n}$, the monotonic tree is better than the random tree. If some of the p_i are large, the savings can be quite substantial.

To demonstrate this, we first consider a set of probabilities satisfying the geometric distribution, $p_i = \frac{r^i}{R}$, $r < 1$, $1 \leq i \leq n$, where $R = \frac{r-r^{n+1}}{1-r}$ is a normalizing constant. Substituting this into the formula for the cost of a monotonic tree, we get

$$2 \sum_{i=1}^{n} H_i \left(\frac{r^i}{R}\right) - 1 = \frac{2}{R} \sum_{i=1}^{n} r^i \sum_{j=1}^{i} \frac{1}{j} - 1$$

$$= \frac{2}{R} \sum_{j=1}^{n} \frac{1}{j} \sum_{i=j}^{n} r^i - 1 = \frac{2}{R} \sum_{j=1}^{n} \frac{1}{j} \frac{r^j - r^{n+1}}{1-r} - 1$$

$$\frac{2}{r-r^{n+1}} \left[\sum_{i=1}^{n} \frac{r^j}{j} - r^{n+1} \sum_{j=1}^{n} \frac{1}{j} \right] - 1$$

If n is large, this is approximately

$$\frac{2}{r} \ln\left(\frac{1}{1-r}\right) - 1, \text{ a constant independent of } n.$$

If the probabilities satisfy Zipf's Law, the cost is

$$2\left[\sum_{i=1}^{n} H_i \cdot \frac{1}{iH_n} \right] - 1 = \frac{2}{H_n} \sum_{i=1}^{n} \frac{H_i}{i} - 1$$

$$= \frac{2}{H_n} \cdot \frac{1}{2} \left(H_n^2 - H_n^{(2)}\right) - 1$$

where

$$H_n^{(2)} = \sum_{i=1}^{n} \frac{1}{i^2} < \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

Thus, for large n, the cost is approximately H_n, which is half of the cost of the random tree. For both distributions the monotonic tree gives significant gains over the random tree.

We now consider a method of selecting the key probabilities that has been studied by Nievergelt and Wong [19]. Here we are given a probability density, f(x), and the key probabilities are chosen with respect to that density. It is necessary to drop the requirement that our choices must sum to one, so instead of probabilities, we must consider key weights. The cost of a tree is now $\sum_{i=1}^{n} w_i \ell_i$ where w_i is the weight of k_i.

We now need two standard definitions

Define : $E(f) = \int_{-\infty}^{\infty} xf(x)dx$, the mean of f(x).

Define : $F(x) = \int_{-\infty}^{x} f(y)dy$, the distribution function of f(x)

F (x) is the probability a number chosen according to the density function is less than or equal to x.

The following theorem defines the cost of a monotonic tree for an arbitrary density function.

Theorem : Given n keys ($k_1 < k_2 < \ldots < k_n$) and a density function f(x), if the weights of the keys are independently chosen from this density function, the expected cost of the resulting monotonic tree is

$$2E(f)[nH_{n-1} - (\frac{n}{2} - 1)] - 2\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)F(y)^i \, dy$$

Proof: As before, we have

$$E(\text{Cost}) = \sum_{i=1}^{n} E(w_i(1 + \sum_{j \neq i} A_{ji})) = nE(f) + \sum_{i=1}^{n} \sum_{j \neq i} E(w_i A_{ji})$$

where w_i is the weight chosen for k_i

and $A_{ji} = \begin{cases} 1 \text{ if } k_j \text{ is an ancestor of } k_i \\ 0 \text{ if not} \end{cases}$

Note that w_i and A_{ji} are not independent.

$$E(w_i A_{ji}) = \int_{-\infty}^{\infty} y \, \text{Prob}(w_i A_{ji} = y) dy$$

A_{ji} can just equal 0 or 1, and if $A_{ji} = 0$ the only y having nonzero probability is $y = 0$. Since this will be multiplied by $y = 0$, the case with A_{ji} can be ignored and

$$E(w_i A_{ji}) = \int_{-\infty}^{\infty} y \, \text{Prob}(w_i = y \text{ and } A_{ji} = 1) dy$$

To determine $\text{Prob}(w_i = y \text{ and } A_{ji} = 1)$ we note that k_j will be an ancestor of k_i if and only if $w_j > w_i$ and w_j is greater than the weight of any key between k_i and k_j in the ordering on the keys. The probability that $w_i = y$ is $f(y)dy$. We then chose an $x \geq y$ for w_j. Any specific x is chosen with probability $f(x)dx$. For this x, we must chose the $|i-j|-1=m$ keys between k_i and k_j to have weight less than or equal to x. The probability for this is $F(x)^m$. The product of these must be integrated over $x \geq y$, giving

$$\text{Prob}(w_i = y \text{ and } A_{ji} = 1) = \int_y^\infty f(y)\, f(x) F(x)^m \, dx dy$$

$$= f(y) \frac{1-F(y)^{m+1}}{m+1} \quad , \quad \text{since } \frac{dF(x)}{dx} = f(x) \text{ and } F(\infty) = 1.$$

Then

$$E(w_i A_{ji}) = \int_{-\infty}^\infty yf(y) \left[\frac{1-F(y)^{m+1}}{m+1}\right] dy.$$

Note that this quantity depends only on m, and not the values of i and j. Since there are 2(n-m-1) distinct ordered (i,j) pairs having a given value of m,

$$E(\text{Cost}) = nE(f) + \sum_{m=0}^{n-2} 2(n-m-1) \int_{-\infty}^\infty yf(y) \left[\frac{1-F(y)^{m+1}}{m+1}\right] dy$$

$$= nE(f) + 2\sum_{m=0}^{n-2} \frac{n-(m+1)}{m+1} \int_{-\infty}^\infty yf(y)dy - 2\sum_{m=0}^{n-2} \frac{n-m-1}{m+1} \int_{-\infty}^\infty yf(y)F(y)^{m+1} \, dy$$

$$= 2E(f)[nH_{n-1} - (\tfrac{n}{2}-1)] - 2\sum_{m=1}^{n-1} \frac{n-m}{m} \int_{-\infty}^\infty yf(y)F(y)^m \, dy$$

Nievergelt and Wong give us two measures to which we can compare this cost.

Theorem (Nievergelt and Wong [19]): The cost of the optimal tree whose weights are chosen according to f(x) is E(f) n log n + O(n).

Theorem (Nievergelt and Wong [19]): The cost of a random tree whose weights are chosen according to $f(x)$ is $(2 \ln 2)\, E(f)\, n \log n + O(n)$.

Nievergelt and Wong also considered choosing the weights for a monotonic tree from a uniform distribution. The resulting cost was $(2 \ln 2)\, E(f)\, n \log n + O(n)$, asymptotically equal to that of the random tree. They conjectured that this held for *any* probability distribution. We now show this conjecture to be true. I am grateful to D. L. Burkholder for the proof of the following lemma:

Lemma: For any density function f with a finite mean,

$$\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)F^i(y)dy = o(n \log n).$$

Proof: First note that for any $i \geq 1$,

$$\left|yf(y)F^i(y)\right| \leq \left|yf(y)\right| .$$

Hence Lebesgue's Dominated Convergence Theorem (see [21]) applies and we have

$$\lim_{i\to\infty} \int_{-\infty}^{\infty} yf(y)F^i(y)dy = \int_{-\infty}^{\infty} yf(y) \lim_{i\to\infty} F^i(y)dy = 0$$

since

$$\lim_{i\to\infty} F^i(y) = 0 \text{ if } F(y) < 1 = 0 \text{ and } f(y) = 0 \text{ if } F(y) = 1.$$

Now,

$$\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)\, F^i(y)dy$$

$$< n \sum_{i=1}^{n-1} \frac{1}{i} \int_{-\infty}^{\infty} yf(y)\, F^i(y)dy \qquad (1)$$

We now choose N such that

$$\int_{-\infty}^{\infty} yf(y)\, F^i(y)dy < \varepsilon \text{ for } i \geq N.$$

Putting this in (1) gives

$$< n \sum_{i=1}^{N-1} \frac{1}{i} \int_{-\infty}^{\infty} yf(y)\, F^i(y)dy + n \sum_{i=N}^{n-1} \frac{\varepsilon}{i}$$

$$< n \sum_{i=1}^{N-1} \frac{E(f)}{i} + n \sum_{i=N}^{n-1} \frac{\varepsilon}{i}$$

Since $H_x < \ell n\, x + 1$, we have

$$< n(\ln(N-1) + 1)\, E(f) + n\varepsilon(\ln(n-1) + 1)$$

Therefore

$$\frac{n(\ln(N-1) + 1)\, E(f) + n\varepsilon(\ln(n-1) + 1)}{n \log n}$$

$$= \frac{(\ln(N-1) + 1)\, E(f)}{\log n} + \frac{\varepsilon(\ln(n-1) + 1}{(\log e)(\ln n)}$$

$$= \frac{(\ln(N-1)+1)\ E(f)}{\log n} + \frac{\varepsilon}{\log e} + \frac{\varepsilon}{(\log e)(\ln n)}$$

We can make the first and third terms arbitrarily small (say, less than ε) by choosing n sufficiently large. Therefore,

$$\frac{\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)\ F^i(y)dy}{n \log n} < (2 + \frac{1}{\log e})\ \varepsilon$$

for $n > N'$. Therefore the limit of this ratio is zero as $n \to \infty$ and the lemma is proved. □

Theorem: If n keys have their weights chosen according to any density function with finite mean, the expected cost of a monotonic tree is $(2 \ln 2)E(f)\ n \log n + O(n)$, asymptotically equal to the cost of a tree built from a random insertion sequence.

Proof: The cost of a monotonic tree is

$$2E(f)[nH_{n-1} - (\frac{n}{2} - 1)] - 2\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)\ F^i(y)\ dy$$

The first term is asymptotically equal to $2E(f)\ n \ln n = E(f)\ n \log n$. The final term was shown by the lemma to be $o(n \log n)$. Hence the asymptotic cost is $(2 \ln 2)\ E(f)\ n \log n$.

We now show that the cost of a monotonic tree is less than or equal to that of a random tree, proving that the cost of a monotonic tree equals $(2 \ln 2)E(f)\ n \log n + O(n)$.

The method we are using to select key weights choses n weights independently from a density function. An equivalent method first selects a set of n weights from an n-dimensional density function. This function is constructed so that the probability of choosing a given set equals the probability of obtaining it (in any order) from n selections from the original function. We then choose a permutation of the set.

Now consider any set. We have already studied the case where the key probabilities (easily generalized to include key weights) were selected from a set and found the expected cost of a monotonic tree to be less than or equal to that of a random tree. Since this holds for every set in the n-dimensional probability density, the theorem is proved. □

Finally, we cite the results of a simulation run by Walker and Gottlieb [14] that showed the performance of monotonic trees to be poor. They state that although these poor results are partially explained by the fact that the leaf weights cannot influence the structure of the tree, even the tests with all leaf weights equal to zero did not produce acceptable nearly optimum trees.

Indeed the majority of the results concerning monotonic trees are quite discouraging. This method performs well only when we are quaranteed that the key probabilities will differ significantly from a uniform distribution (i.e., have low entropy). If this is not the case (as in the situation described by Nievergelt and Wong), the performance is asymptotically the same as randomly built trees.

3.3 Cost Balanced Trees

The previous methods have focused on the fact that a rotation moves a certain node up in the tree, ignoring the fact that it also disturbs two (possibly large) subtrees. The method of cost balancing considers the entire tree. We do a rotation only when it appears to be profitable, that is, when the number of accesses to the nodes that will move up exceeds the number to those that will move down.

This method has the advantage that it is possible to do the rebalancing during the search for the requested key since we know in which subtree it lies. For example in Figure 3.3.1, we perform a rotation to promote A, if $w(A) + w(S_1) > w(B) + w(C) + w(S_3) + w(S_4)$. We promote C if $w(C) + w(S_4) > w(B) + w(A) + w(S_1) + w(S_2)$. Here, $w(A)$ is the number of times A has been requested, and $w(S_1)$ is the number of times any node in S_1 has been requested. All this information is available at node B, and any rebalancing can be done there.

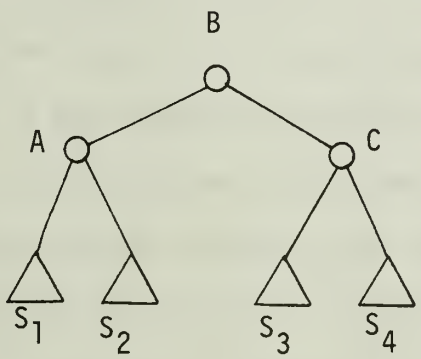


Figure 3.3.1

Another advantage of this rule is that the leaf weights play a rule in the balancing of the tree. Since a rotation that promotes the nodes in subtree S_1 also must promote the leaves, the "weight" of S_1 must be the number of accesses to both the nodes and the leaves of S_1. If the weight of the leaves is considerable, this is a significant advantage over previous rules, all of which ignored accesses to leaves.

However, balancing at one node may cause other nodes to become unbalanced. (See below).

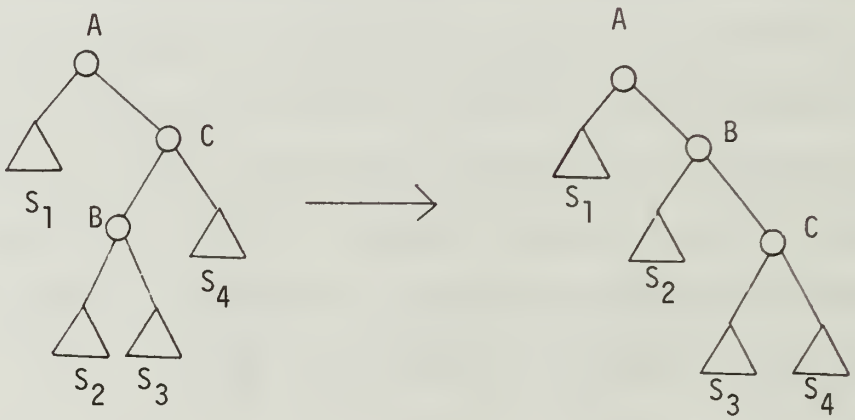


Figure 3.3.2

Here, both node A and node C may require rebalancing. (However, no rebalancing would be required at node A if node C had been its left son). An attempt to correct these imbalances (and all the imbalances resulting from the corrections) could be quite costly. A more reasonable policy is to ignore the imbalances and rebalance at a later request when the search path passes through the unbalanced node.

Tables 3.3.1 and 3.3.2 compare these two rules. The total rotation rule (which corrects all imbalances) has a slightly lower cost.

Table 3.3.1

The Performance of the Limited Single Rotation (LSR) Rule

		ZIPF'S LAW					
	English Letters	#1	#2	#3	#4	#5	Average
Random Cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal Cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
LSR Cost	3.44	4.33	4.14	4.46	4.28	4.20	4.28
Increase Over Optimum	3.55%	5.46%	5.31%	7.29%	5.42%	5.98%	5.89%
Average Number of Rotations/Request	.111	.199	.204	.199	.197	.200	.200
Average Over the Last 100 Requests		.033	.041	.040	.039	.034	.038

See Table 3.1.1 for explanation.

Table 3.3.2

The Performance of the Total Single Rotation (TSR) Rule

		ZIPF'S LAW					
	English Letters	#1	#2	#3	#4	#5	Average
Random Cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal Cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
TSR Cost	3.41	4.33	4.11	4.41	4.22	4.17	4.25
Increase Over Optimum	2.93%	5.57%	4.57%	6.02%	3.92%	5.27%	5.07%
Average Number of Rotations/Request	.118	.220	.219	.217	.209	.213	.215
Average Over the Last 100 Rotations		.040	.048	.044	.036	.036	.041

See Table 3.1.1 for explanation.

It gives an increase of 5.07% over the optimal cost (as compared with 5.89 percent for the limited rotation rule) with a suprisingly small increase in the number of rotations required (an average of .215 per request as compared with .200). However, there is much more overhead associated with a rotation in the total rotation rule. Since imbalances can propogate throughout the tree, either a pointer to a node's father must be maintained or we must stack the nodes encountered during the search for the requested key.

These tables also show how much work the rules do after many requests. We consider the last 100 requests out of 500 in the simulation. The limited rotation rule does an average of .038 rotations per requested during this period, or approximately one rotation every 27 requests. The total rotation rule averages .041 rotations per request, or one rotation every 24 requests.

A weakness of these rotation rules is that they do not consider the "inside" subtrees (the right subtree of a node's left son, or the left subtree of its right son, see Figure 3.3.3).

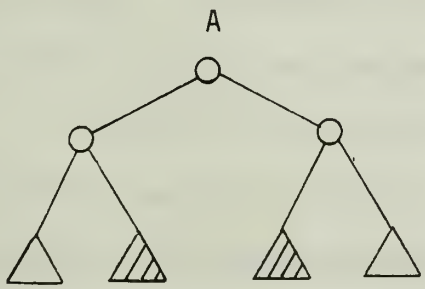


Figure 3.3.3 The inside subtrees of node A are darkened.

A rotation can promote either exterior subtree, but the interior subtrees remain at the same level. This can lead to very poor trees that are still "stable" in that no rotation can be performed. Figure 3.3.4 shows an example. This tree is stable as long as the weight of a node is less than or equal to that of his father.

Figure 3.3.4

While the worst case performance on such a distribution is quite bad, a simulation suggests the average case is acceptable. The probability distribution was $p_1 = \frac{50}{1275}$, $p_2 = \frac{48}{1275}$, ... , $p_{25} = \frac{2}{1275}$, $p_{26} = \frac{1}{1275}$, $p_{27} = \frac{3}{1275}$ ... $p_{50} = \frac{49}{1275}$. The tree shown in Figure 3.3.4 is stable for this probability distribution. Yet, after 500 requests the limited rotation rule reduced the cost to 4.7593, a mere 3.06 percent increase over the optimal cost of 4.6180.

The limited single rotation rule has several desirable features. The necessary rotations can be performed during the search for the requested key. The performance for the distributions we considered was good, within 5.89 percent of the optimal. After an initial

period that reorganizes the tree, the rule required a rotation approximately every 27 requests, a very low maintenance cost.

The total single rotation rule has little more to offer. It decreases the cost to within 5.07 percent of the optimum, and surprisingly does only slightly more rotations. However the overhead required by this rule to allow changes to propogate throughout the tree is not justified by the relatively small decrease in cost, making the limited rotation rule a better choice.

3.4 Double Rotations

A transformation (called a "double rotation") that allows the promotion of inside subtrees is shown below.

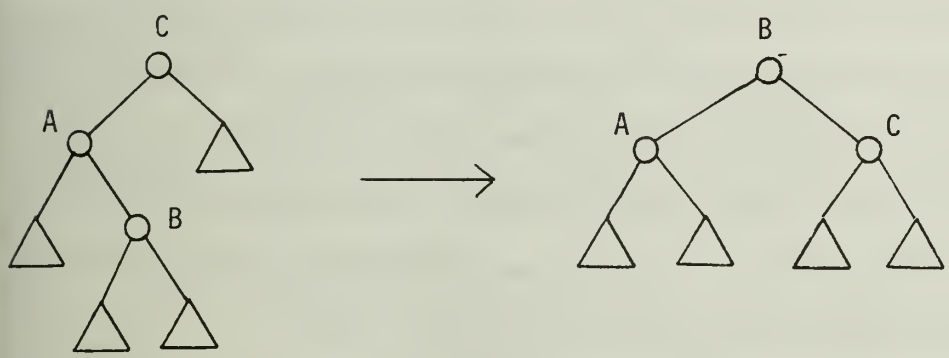


Figure 3.4.1 A double rotation.

A rule that uses both single and double rotations has several advantages over a rule that is limited to single rotations. First, such a rule will always be able to promote a requested node if it is profitable to do so. Single rotations can be used to promote nodes in the outside subtrees and double rotations for those in the inside subtrees.

A double rotation actually consists of two successive single rotations, each promoting node B one level. The double rotation has an advantage when doing both of these rotations will reduce the cost, but doing only the first will not. A rule that is restricted to performing single rotations will check if the first rotation can be done. Since it cannot be, the tree is left unchanged, and the second rotation is not considered. A rule which also considers double rotations will be able to reduce the cost in this situation.

Table 3.4.1 shows the cost of a rule that uses both single and double rotations. For the distribution considered, this method averaged within 3.84 percent of the optimal cost. The total number of rotations required per request (counting both single and double rotations) is .208, which is very close to the averages for the limited rotation rule (.200) and the total rotation rule (.215).

However, after many requests, fewer rotations are required than for either single rotation rule. In fact, the average over the last 100 requests was .027 single rotations (one every 36 requests) and .008 double rotations (one every 129 requests).

Table 3.4.1

The Performance of the Double Rotation (DR) Rule

		ZIPF'S LAW					
	English Letters	#1	#2	#3	#4	#5	Average
Random Cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal Cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
DR Cost	3.40	4.29	4.06	4.32	4.23	4.09	4.20
Increase Over Optimum	2.35%	4.62%	3.45%	3.80%	4.11%	3.22%	3.84%
Average Number of Single Rotations/ Request	.074	.129	.129	.127	.126	.126	.127
Average Over Last 100 Requests		.029	.027	.025	.028	.028	.027
Average Numbers of Double Rotations/ Request	.037	.082	.081	.082	.079	.080	.081
Average Over Last 100 Requests		.007	.007	.008	.007	.009	.008

See Table 3.1.1 for explanation.

These results are supported by a simulation run by Baer [20]. He assumed all keys to be equally likely and found the cost of the double rotation rule to range from 1.2 percent to 3.6 percent of the optimum. This cost is lower than that we obtained because more requests were made in Baer's simulation. In addition, his trees have fewer nodes. This would also explain the lower cost since the cost of smaller trees tends to be closer to the optimum (see Table 2.9.1, compare the cost for the English letters (26 nodes) with the others (100 nodes)).

Baer also gives statistics on the number of rotations done by this rule. Again his results agree with ours. His most extensive simulation (85,000 requests) required 23 rotations for the first 850 requests (one every 37 requests). The next 7650 requests caused 6 rotations (one every 1,275 requests), and the final 76,500 also caused 6 requests (on every 12,750 requests). These results indicate that the cost of "maintaining" the tree might be extremely low.

Bruno and Coffman [12] have considered an extension of this rule that can promote a node any number of levels by using a sequence of rotations. They, however, were concerned with an algorithm to build a nearly optimal tree from a set of known key probabilities and used this set of transformations to reduce the cost of the initial tree. Every final tree in their simulation was within 5 percent of the optimum, and the average was within 2.6 percent.

This suggests further rules, where we consider promoting the requested node i levels for i = 1, 2, ..., k, where k is a parameter of

the rule. Note that the single rotation rules have k = 1, and the double rotation rule has k = 2. Increasing k will increase the work the rule must do, but will result in decreased retrieval times. The results of Bruno and Coffman suggest that the retrieval time will not be greatly improved by increasing k beyond 2, while the increase in the complexity of the algorithm to execute the rule would be substantial.

3.5 Summary and Conclusion

We have examined several methods for dynamically altering binary search trees to decrease their access time. The first two methods were analogs of the linked list case : the move to root rule and the move up one rule. The move to root rule was shown to be identical to the first request rule (analogous to the case of the linked list) and a formula for the cost was derived. Calculations showed this method to be an improvement over a tree built by a random sequence of insertions. The analogy breaks down when we consider the move up one rule; it is often outperformed by the move to root rule. A simulation showed the move to root rule to have lower average cost than the move up one rule (38 percent of the optimum compared with 45 percent), indicating that of these two rules, the move to root rule would be the superior choice. However, these rules should be used only if we cannot associate a counter with each key. If we can, the following rules will give better performance.

We next considered rules that use counters. The first of these was the monotonic tree rule. Its performance was found to be

disappointing. Melhorne [13] has shown that the ratio of the cost of a monotonic tree to that of the optimal tree may be as high as $n/(4 \log n)$, for a tree of n nodes. If the weights of the nodes are chosen according to a probability density function (a case considered by Nievergelt and Wong [19]) the performance is asymptotically the same as a random tree for any probability distribution. A simulation by Walker and Gottlieb [14] also confirms the poor performance of this method. Only if we assume the probabilities are chosen from a fixed set (guaranteeing they will be "spread out") does this method significantly improve over the cost of a random tree. A formula was derived for this case, and significant decreases were obtained for Zipf's Law and the geometric distribution. This assumption was also true in the simulation we discussed. It showed the cost of this method to average within 15 percent of the optimal cost.

We then discussed the most promising methods. Simulations showed that the cost of the limited single rotation rule averaged within 5.89 percent of the optimum. However, the worst case performance was very bad, resulting in the tree shown in Figure 3.3.4. The total single rotation rule reduced the average cost to 5.07 percent of the optimum. However, since this rule requires much more overhead, this small decrease in cost does not justify its use.

Finally, we considered the double rotation rule. Its cost averaged approximately 3.84 percent of the optimum. Though this rule must check for both single and double rotations, it averages a rotation every 36 requests and a double rotation every 129 after the initial period of reorganization. Compared with the limited single rotation

rule (one rotation every 27 requests), the double rotation rule does less work after the initial period. It then appears to be the best choice of the counter rules.

4. CONCLUSION

The purpose of this thesis was to examine various heuristics that dynamically alter data structures by moving frequently accessed keys near the "top" of the data structure.

The first data structure we considered was the linked list. If relatively few requests (compared to the number of keys) are anticipated, the fast convergence of the move to front rule (nearly as fast as the optimum) makes it the best choice. If many requests are expected, the transposition rule gives the best performance because its asymptotic cost is close (10 percent) to the optimum. For an intermediate number of requests, the first request/transposition rule combines both of these features with a small additional overhead, making it the best choice in this case.

If space is available for counters, there are much better rules. If enough space is available so that the counters will never overflow, the frequency count rule should be used. If this is not the case, the limited difference rule uses whatever space can be spared and gives nearly optimal results for even a small number of bits.

The second data structure we considered was the binary search tree. Here we found the move to root rule to give the best performance of the rules that do not use counters, approximately 38 percent of the optimum. If counters are available, the double rotation rule appears to be best. Its performance averages 3.84 percent of the optimum, and after a period of initial organization of the tree, it is

very inexpensive to execute. On the average, a single rotation is done every 36 requests, and a double rotation every 129 requests.

The methods we have considered are simple and inexpensive to execute. In addition, they significantly reduce the average access time over data structures in which the keys are randomly arranged; in some cases these methods keep the structure very close to the one of optimum cost.

APPENDIX

We will make great use of Markov chains, so a summary of their important properties is required. These can all be found in [1].

To define a Markov chain, we first consider a set S of states and a sequence $\{x_n, n=0,1,\ldots\}$ of random variables which take their values from S. The value x_n tells us which state the chain is "in" at time n.

In addition, the Markov property must be satisfied. This is $\text{Prob}\{x_{n+1} = S_{n+1} | x_0 = S_0,\ldots,x_n = S_n\} = \text{Prob}\{x_{n+1} = S_{n+1} | x_n = S_n\}$, which says that the probability of being in a given state depends only on the previous state, and not any before that. We then define the probabilities $\text{Prob}\{x_{n+1} = j | x_n = i\}$ (or just $P(i,j)$) as the <u>transition probabilities</u> of the chain and can form a matrix whose $(i,j)^{th}$ element is $\text{Prob}\{x_{n+1} = j | x_n = i\}$. This is called the transition matrix P. If we are given a probability distribution $\bar{x}$ over the states, then $\bar{x}P$ gives us the probability distribution at the next time step.

This defines the basic idea of Markov chains. Several more definitions are required.

<u>Definition</u> : A state x <u>leads to</u> a state y if there exists a sequence of states $x_1,\ldots,x_n$ such that

$$P(x,x_1)\ P(x_1,x_2)\ \ldots\ P(x_n,y) > 0.$$

<u>Definition</u> : A set C of states is <u>closed</u> if no state in C leads to a state outside of C.

Definition: A closed set C is irreducible (also called ergodic) if x leads to y for all choices of x and y in C.

Most of the chains we are dealing with will be irreducible. That is the set of all states, S, will be irreducible. S is then closed since there are no states outside of S. For these chains, it will be clear that some sequence of requests can be designed to cause any state to lead to any other state.

Definition: Define the period of a state x as the greatest common divisor (g.c.d.) of the set $\{n \geq 1 : P^n(x,x) > 0\}$. It can be shown that all states have equal periods and this is defined as the period of this chain. A chain with a period of 1 is called aperiodic.

Nearly all of the chains we deal with will be aperiodic. If the top element in a configuration is requested, none of these schemes will alter the configuration and hence we will have $P^1(x,x) > 0$. Hence 1 is in the set $\{n \geq 1 : P^n(x,x) > 0\}$ so the g.c.d. must be 1 and the chain must be aperiodic.

Definition: A steady state distribution (also called a stationary distribution is defined as any probability distribution $\bar{x}$ over the states such that $\bar{x}P = \bar{x}$.

Note that we can easily determine this distribution by solving the system $\bar{x}(P-I) = 0$. The following theorem shows that this distribution tells us the asymptotic behavior of the chain.

Theorem: Any closed and irreducible chain with a finite number of states has a unique steady state distribution. If the chain is aperiodic, it approaches the steady state distribution from any initial distribution. If the chain has period d, then for $0 \leq i < d$, the chain $x_i, x_{i+d}, x_{i+2d} \ldots$ has a unique steady state distribution and approaches it.

Another useful theorem that characterizes asymptotic behavior is the ergodic theorem.

Theorem [Ergodic Theorem]: Let $N_t(s)$ be a random variable denoting the number of times the chain has been in state s after t transitions. Suppose that s has steady state probability p(s). Then if the chain is closed and irreducible (ergodic)

$$\lim_{t\to\infty} \frac{N_t(s)}{t} = p(s).$$

Note the ergodic theorem holds for both periodic and aperiodic chains.

The chains we are dealing with have a cost associated with each state. Let the probability of being in state s at time t be $p_t(s)$, and suppose s has steady state probability p(s) and cost c(s). Finally define the cost of the chain at time t ($COST_t$) as $\sum_{s\in S} p_t(s)c(s)$.

For an aperiodic irreducible chain, we have $\lim_{t\to\infty} p_t(s) = p(s)$. Hence $\lim_{t\to\infty} COST_t = \sum_{s\in S} p(s)c(s)$.

For any irreducible chain, we can use the Ergodic Theorem to show

$\lim_{t\to\infty} \frac{1}{t} \sum_{i=1}^{t} COST_i = \sum_{s\in S} (s)c(s)$. Therefore $\sum_{s\in S} p(s)c(s)$ determines the asymptotic cost of the chain. We also use the following theorem.

Theorem: Let c_i be random variables denoting the cost of the state the chain is in at time i. Then for any irreducible chain,

$$\lim_{t\to\infty} E\left(\frac{c_1+c_2+\ldots+c_t}{t}\right) = \sum_{s\in S} p(s)c(s)$$

$$\lim_{t\to\infty} VAR\left(\frac{c_1+c_2+\ldots+c_t}{t}\right) = 0.$$

Proof: To prove the first statement, note $E(c_i) = COST_i$. Then

$$\lim_{t\to\infty} E\left(\frac{c_1+c_2+\ldots+c_t}{t}\right) = \lim_{t\to\infty} \frac{\sum_{i=1}^{t} COST_i}{t} = \sum_{s\in S} p(s)c(s).$$

To prove the second statement,

$$\lim_{t\to\infty} VAR\left(\frac{c_1+c_2+\ldots+c_t}{t}\right)$$

$$= \lim_{t\to\infty} E\left[\left(\frac{c_1+c_2+\ldots+c_t}{t}\right)^2\right] - \left(\sum_{s\in S} p(s)c(s)\right)^2$$

$$= E\left[\left(\lim_{t\to\infty} \frac{c_1+c_2+\ldots+c_t}{t}\right)^2\right] - \left(\sum_{s\in S} p(s)c(s)\right)^2 \qquad (1)$$

We now determine $\lim_{t\to\infty} \left(\frac{c_1+c_2+\ldots+c_t}{t}\right)$. Let $N_t(s)$ be the number of times state s is visited in t transitions. Then $c_1+c_2+\ldots+c_t$ equals the number of times we visit each state times its cost summed over all states.

Therefore

$$\lim_{t\to\infty}\left(\frac{c_1+c_2+\ldots+c_t}{t}\right) = \lim_{t\to\infty}\frac{\sum_{s\in S} N_t(s)c(s)}{t}$$

$$= \sum_{s\in S}\left(\lim_{t\to\infty}\frac{N_t(s)}{t}\right)c(s) = \sum_{s\in S} p(s)c(s)$$

by the Ergodic Theorem. Substituting this into (1) shows the variance is zero. □

Another important question is how quickly the chain approaches steady state. We can tell this from the eigenvalues of the transition matrix. To see this, suppose the n eigenvectors $\bar{y}_1,\ldots,\bar{y}_n$ span the space of all probability distributions. We can then write an initial distribution $\bar{x}_0$ as a linear combination of the $\bar{y}_i$

$$\bar{x}_0 = c_1\bar{y}_1 + \ldots + c_n\bar{y}_n$$

So after t transitions, the distribution will be

$$\bar{x}_0P^t = c_1\lambda_1^t\bar{y}_1 + \ldots + c_n\lambda_n^t\bar{y}_n.$$

Since the chain has a stationary distribution, there is some $\bar{x}$ such that $\bar{x}P = \bar{x}$. Hence $\bar{x}$ is an eigenvector with eigenvalue 1. If the chain is closed, irreducible and aperiodic, we can show that all other eigenvalues have modulus strictly less than 1. If we suppose λ_1 is the eigenvalue equal to 1, we get $\bar{x}_0P^t = \bar{x} + \lambda_2^t c_2\bar{y}_2 + \lambda_3^t c_3\bar{y}_3 + \ldots + \lambda_n^t c_n\bar{y}_n$. Then as $t \to \infty$, $\bar{x}_0P \to \bar{x}$, and the rate of convergence is limited by the size of the other n-1 eigenvalues and eigenvectors.

REFERENCES

[1] Hoel, P. B., S. C. Port, and C. J. Stone, Introduction to Stochastic Processes, Houghton Mifflin Company, Boston, 1972.

[2] Rivest, R. L., "On Self Organizing Sequential Search Heuristics," CACM, 19 (1976), 63-67.

[3] Knuth, D. E., The Art of Computer Programming, Vol. 3, Addison-Wesley Publishing Co., Reading, Mass., 1973.

[4] Yao, A. C., Personal communication.

[5] Kahn, D., The Codebreakers, Macmillan Company, New York, 1967.

[6] Kučera, H. and W. N. Francis, Computational Analysis of Present-Day American English, Brown University Press, Providence, R.I., 1967.

[7] Burville, P. J. and Kingman, J.F.C., "On a Model for Storage and Search," J. Appl. Prob., 10 (1973), 697-701.

[8] Hendricks, W. J., "The Stationary Distribution of an Interesting Markov Chain," J. Appl. Prob., 9 (1972), 231-233.

[9] Hendricks, W. J., "An Extension of a Theorem Concerning an Interesting Markov Chain," J. Appl. Prob., 10 (1973), 886-890.

[10] McCabe, J., "On Serial Files with Relocatable Records," Operations Res., 12 (1965), 609-618.

[11] Knuth, D. E., "Optimum Binary Search Trees," Acta Informatica, 1 (1971), 14-25.

[12] Bruno, J. and E. G. Coffman, "Nearly Optimal Binary Search Trees," Proc. IFIP Congress 71 (1971).

[13] Melhorne, K., "Nearly Optimal Binary Search Trees," Acta Informatica, 5 (1975), 287-295.

[14] Walker, W. A. and C. C. Gottlieb, "A Top-Down Algorithm for Constructing Nearly Optimal Lexicographic Trees," in Graph Theory and Computing (ed. R. C. Reid), Academic Press (1972), 303-323.

[15] Adel'son-Vel'skii, G. M. and E. M. Landis, Dokl. Akad. Nauk SSSR 146 (1962), 263-266; English translation in Soviet Math 3, 1259-1263.

[16] Nievergelt, J. and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM J. Comput. 2 (1973), 33-43.

[17] Williams, J.W.J., Algorithm 232 - Heapsort., CACM 7 (1964), 347-348.

[18] Hibbard, T. N., "Some Combinatorial Properties of Certain Trees with Application to Searching and Sorting," JACM 9 (1962), 16-17.

[19] Nievergelt, J. and Wong, C. K., "On Binary Search Trees," Information Processing 71, vol. 1, North Holland, Amsterdam, pp. 91-98.

[20] Baer, J. L., "Weight Balanced Trees," National Computer Conference 1975, pp. 467-472.

[21] Rudin, W., Principles of Mathematical Analysis, McGraw-Hill, New York, 1976, p. 321.

VITA

James Richard Bitner was born August 2, 1953, in Minneapolis, Minnesota. He attended the University of Illinois at Urbana-Champaign, receiving a B.S. in Mathematics and Computer Science (June 1973) with highest honors and distinction in Mathematics and Computer Science. He continued at the University of Illinois for graduate study, receiving an M.S. (December 1974) in Computer Science. During this time, he was employed as a research assistant for Drs. C. L. Liu and E. M. Reingold. He is also a member of Phi Beta Kappa and Phi Kappa Phi. The titles of his published articles are: "Use of Macros in Backtrack Programming," "Tiling 5n x 12 Rectangles with Y-Pentominos" and "Efficient Generation of the Binary Reflected Gray Code and Its Applications."

LIOGRAPHIC DATA ET	1. Report No. UIUCDCS-R-76-818	2.	3. Recipient's Accession No.
tle and Subtitle EURISTICS THAT DYNAMICALLY ALTER DATA STRUCTURES TO CREASE THEIR ACCESS TIME			5. Report Date July 1976 6.
uthor(s) James R. Bitner			8. Performing Organization Rept. No.
erforming Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF GJ-41538
ponsoring Organization Name and Address National Science Foundation Washington, D.C.			13. Type of Report & Period Covered 14.

Supplementary Notes

Abstracts
In evaluating the access times of data structures, we often assume that each key is equally likely to be requested. This is seldom the case in practice: some keys will be requested more often than others. The data structures considered are lists and trees, and access times can be substantially reduced in these data structures by the use of several simple heuristics that cause the more frequently accessed key to move to the "top" of the data structure. These methods are analyzed and compared.

Key Words and Document Analysis. 17a. Descriptors
Data structures, linked lists, binary search trees, access times.

Identifiers/Open-Ended Terms

COSATI Field/Group

Availability Statement

19. Security Class (This Report) UNCLASSIFIED
21. No. of Pages 135
20. Security Class (This Page) UNCLASSIFIED
22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71